NRL Memorandum Report 5502
December 31, 1984

Interface Specifications for the SCR (A-7E) Extended Computer Module

by

David Lorge Parnas
Kathryn Heninger Britton
David M. Weiss
Paul C. Clements

Naval Research Laboratory
Washington, D. C.

EC.INTRO: Introduction

1. Overview

The Extended Computer (EC) is a computing machine partially implemented in software. It was designed as part of the Software Cost Reduction (SCR) project at the Naval Research Laboratory. The design goals are 1) code portability, 2) abstraction from computer hardware idiosyncracies, 3) more easily understood code, and 4) sharing of solutions to common machine dependent coding problems. The Extended Computer is designed to be efficiently implemented on avionics computers such as the IBM 4PI TC-2.

The Extended Computer has the following features:

- *Extensible addressing*. There is no syntactic limit to the amount of memory that can be addressed. The actual memory size is a parameter that is set at system-generation time.
- *Uniform data access*. Hardware addressing techniques, such as use of base and link registers, are hidden from programmers.
- *Uniform subprogram access*. All subroutines are invoked in a uniform manner; linkage mechanisms are hidden from users.
- *Uniform input/output.* Variations in I/O operations are hidden. All input (output) data items are read (written) using the same statements.
- *Uniform event signalling.* The difference between hardware interrupts and software-detected events is hidden. All interrupt handling is hidden.
- *Data types*. Data types representing reals, bitstrings, and time intervals are provided together with the necessary conversion functions. Data representations are hidden. Hardware arithmetic and bitstring operations are hidden.
- *Parallel processes*. Programs can be written as a set of cooperating sequential processes. The number of hardware processors and their scheduling are hidden.
- *State control*. Computer state transitions among various states (including off, operating, and failed) are signalled to the user programs. The mechanics of state transitions are hidden.
- *Built-in test*. Diagnostic programs to test the integrity of memory and the correct operation of the hardware are built-in. The tests and evaluation criteria are hidden.
- *Exception handling*. Both a development version, with extensive checks for programming errors, and a production version are available. Programs that cause no undesired events [WUER76] on the development version will compute the same values on both versions. The version can be selected at system-generation time.

The Extended Computer has been designed to hide the interface characteristics of a computer with capabilities similar to those of the IBM 4PI/TC-2. Were the present A-7 computer to be replaced by one with different capabilities, we would shift some responsibilities to/from other parts of the software. For example, if the new computer used an external device for timing, the implementation of the timeint data type would become a part of the device interface modules. Or, if the new computer included a capability for angle implementation, the machine-independent implementation of an angle data type would be replaced by a machine-dependent module that was part of the EC, but with the same interface as the present angle data type. Of course, under such unlikely circumstances, the appropriate documentation (such as [REQ], [MG], and [AT], as well as this document) would be changed to remain consistent with the new hardware. If the EC design were to be used in an application that did not require all of its capabilities, a compatible subset could be used.

We recommend that this procedure be followed by anyone maintaining this system, and by those who are designing other systems using a similar approach.

2. Conventions of this document

This document specifies the user interface to the Extended Computer. The contents, form, and notation are in accordance with the guidelines given in [SO], with the following additions.

- **Events signalled by incrementing a semaphore:** The EC signals events by incrementing semaphores. The semaphores and the event that each represents are listed in a separate section of each submodule interface. The semaphores are built-in (users need not declare them), and are given an initial value of zero at system generation time.

  For each built-in semaphore *S*, there is a built-in (i.e., pre-declared) region *S_R*. The EC signals an event by performing an +UP+ followed by a +DOWN+ on that semaphore, and both operations occur inside the corresponding region. Programmers may use that the name of that region in exclusion relations. See EC.SMPH for information about semaphore operations and EC.PAR.2 for exclusion relations.

- **Optional parameters:** If a parameter is denoted by "I_OPT" in the access program table, it means that the corresponding actual parameter may be omitted in an invocation of that program. No EC program has more than one optional parameter.

3. Input to the Extended Computer

Text input to the EC is partitioned into a non-overlapping sequence of *tokens*. A token can be

- a name (defined in EC.DATA.3);
- a real or bitstring literal (defined in EC.DATA.3);
- one of the seven punctuation characters < > ( ) [ ] :

Tokens may not exceed a certain length, specified by #max token length#; %%token too long%% will be raised if the limit is violated.

A *comment* is defined by the BNF syntax

```
comment           ::= '{' comment_contents '}'
comment_contents  ::= empty | comment_unit comment_contents
comment_unit      ::= most_ascii | comment
most_ascii        ::= any ascii character except '{' or '}'
```

White space is any nonempty sequence of the ASCII characters SP (octal 040), HT (011), NL (012), and FF (014) occurring outside of tokens and comments.

The input is scanned from beginning to end for tokens. White space and comments are discarded but may serve to separate two tokens that would otherwise be read as one. The longest possible tokens are recognized; the shortest possible comments are recognized.

Syntactic input to the EC consists of a sequence of invocations of system- generation-time access programs. The syntax of invocation is given in EC.PGM.3.2.

Violation of any of the above rules will result in the undesired event %%syntax error%%.

The program that produces target machine code from input to this module is called **ect**; its user interface is described in Chapter TT.TRANS of [TT].

CHAPTER 1

EC.DATA: Data Manipulation Facilities

1. Introduction

1.1. Entities

The Extended Computer provides literals, constants, and variables. We refer to these as *entities. Literals* are values appearing in programs. *Constants* have names and values; run-time programs can read the values but not change them. *Variables* have names and values; the values can be read or written by run-time programs. All constants and variables may be accessed from any process of the program. It is possible to declare arrays of variables or constants. An element of an array may be used as an individually declared entity of the same type. Users are given the facility for providing information to the Extended Computer about the relative speeds with which declared entities should be accessed.

1.2. Types

Types are classes of entities. The Extended Computer provides a hierarchy of types; an entity is either a program, numeric, bitstring or pointer. Numeric types are characterized by *range* and *resolution*. Bitstring types are characterized by *length*. The value of a pointer is another entity. Pointer types are characterized by the type of entity to which members of the pointer type may refer. The value of a characteristic for an entity is called an *attribute*. Some entities are allowed to change their type at run-time.

For a particular numeric type, every numeric value between the upper bound and lower bound (inclusive) has a representative in its type. Any representative will differ from its nearest neighbors by no more than the resolution of the type, and no numeric value will differ in value from its representative by more than half the resolution. The values referred to in this specification are the representative values.

For some numeric types, users may require that the representatives include exact multiples of the resolution between the lower and upper bounds, inclusively.

A *type class* is a type that contains entities with different behavior. A *specific type* (also called *spectype*) is a subclass of a type class in which all variables have identical behavior; i.e., they can take on the same set of values and one may perform the same operations on them with the same results. The behavior of the program will not change if two variables of the same specific type are interchanged throughout the program. For each type class, there are any number of specific types.

Figure 1 provides an overview of the EC data types by showing the Extended Computer's type classes. A sub-type is indented beneath its containing types. Terminal nodes (those entries with no sub-types) represent type classes in which specific types may be declared.

The Extended Computer provides three type classes illustrated in Figure1, but not described in this chapter. They are *semaphore, timer,* and *program*, whose operations are described respectively in EC.SMPH, EC.TIMER, and EC.PGM.2.

All	bitstrings		
	programs		
	numerics	reals	integers
			nonintegers
		time intervals	
		semaphores	
		timers	
	pointers	data pointers	
		program pointers	

Figure 1 -- EC Data Types

1.3. Scalar literals

A scalar literal is an unnamed ascon. Formats for writing literal values for a particular typeclass are specified in the type definition for that class in EC.DATA.3. Formats for using literals as operands to access programs are given in EC.DATA.2.4.

Numeric literals will be represented with at least the precision implied by their written representation. Integers will be represented exactly.

2. Interface overview

2.1. Declaration of specific types

Specific types must be declared, given a name, and assigned attributes. The EC allows users to choose among different versions of the implementation for each type; each version is especially efficient for performing certain operations. The versions, and the advantages and disadvantages of each, are specified in Appendix F.

Program	*Parameters*	*Description*	*Undesired events*
++DCL_TYPE++	p1: name; I	name of new type	
	p2: typeclass; I	containing type class	
	p3: attribute; I	attributes of type	
	p4: version; I	implementation version	
			%%name in use%%
			%%inappropriate attributes%%
			%%illegal length%%
			%%malformed attributes%%
			%%range too great%%
			%%version characteristic exceeded%%
			%%res too fine%%
			%%undefined name%%

Effects

A specific type that is a member of type class p2 and implementation version p4 is declared to have identifier p1. All entities and arrays of this specific type will have the attributes given by p3. If p4 is not a version associated with the given type, as specified in Appendix F, then the EC implementation will use an appropriate version of its own choosing. The identifier can be used as the spectype (p2) parameter in calls to ++DCL_ENTITY++, ++DCL_ARRAY++, and ++DCL_TYPE_CLASS++ that follow the declaration.

The Extended Computer gives users a way to define a type class as a !!list!! of specific types, for use in declaring entities.

Program	*Parameters*	*Description*	*Undesired events*
++DCL_TYPE_CLASS++	p1: name; I	name of list	
	p2: type_list; I		
			%%name in use%%
			%%undefined name%%

Effects

The name p1 can be used in place of the !!list!! given in p2.

2.2. Declaration and ranking of data sets

The EC requires users to assign entities to data sets. The user is then allowed to specify a partial ordering on the data sets to determine speed of access to the sets' members. The rankings apply to sections of code. Significant performance improvements are possible if the entities used in a section of code belong to a data set that is highly ranked.

Program	*Parameters*	*Description*	*Undesired events*
++DCL_DATA_SET++	p1: name; I	name of data set	

			%%name in use%%
++RANK_DATA_SET++	p1: data-set-reln; I		
			%%undefined name%%

Effects

++DCL_DATA_SET++ Declares p1 to be the name of a data set, and allows that name to be used as p5 of ++DCL_ENTITY++ and/or p6 of ++DCL_ARRAY++.

++RANK_DATA_SET++ Defines a partial ordering on all data sets; if (A B) is in the !!relation!! given by p1, then data set A has a rank no lower than that of data set B. Data sets not named in p1 have an arbitrary rank lower than any set named in p1. (A special case of this is when the data-set-reln is empty; in this case all data sets are given an arbitrary rank by the EC.) The ranking applies until the next textual occurrence of ++RANK_DATA_SET++. This program's only visible effects are to alter the performance of the user's program, as specified in Appendix G.

2.3. Data declarations

Variables, constants and arrays must be declared before they are used. The declaration must specify the name of the new entity or array, a previously declared specific type (one of the terminal nodes on the tree of Figure 1), whether the entity or array is constant or a variable, and an initial value.

Program	*Parameters*	*Description*	*Undesired events*
++DCL_ENTITY++	p1: name; I	entity name	
	p2: type_list; I	specific types	
	p3: convar; I	when writeable?	
	p4: see below; I	initial value	
	p5: data_set; I_OPT		
		data set name	
			%%literal too big%% %%name in use%% %%undefined name%% %%unknown initial value%% %%untyped literal%% %%varying constant%% %%wrong init value type%%

Parameters

p4 must be given by a constant or a !!typed literal!! or the built-in identifier UNDEF. If p2 contains only one specific type, p4 may also be given by a simple literal. The type of a !!typed literal!! or constant must match one of the specific types named (or referred to) in p2. The value of a simple literal must be in the domain of the type named in p2.

p2 may only contain or refer to more than one specific type if p3=VAR.

Effects

An entity with identifier p1 and initial value p4 is declared. If p4 is given by a constant or !!typed literal!!, p1 assumes its spectype. If p2 contains more than one type, the entity is allowed to subsequently assume the attributes of any of the types named in p2; see the "Operand descriptions" section for more information. If p3=VAR, the entity may be used as a !!destination!! in a subsequent operation. The entities that have been declared may be used as operands in the programs that follow. The entity is assigned to data set p5 (or to none if p5 is omitted). If p4=UNDEF, a value must be assigned to this entity before it is used as an I or IO operand to an EC access program. If p2 is given by a single type that has the EXACT_REP attribute, and p4 is given by a simple literal whose value is an exact integer multiple of the type's resolution, then p4 will be represented exactly as given.

2.3.1. Declaration of arrays

Program	*Parameters*	*Description*	*Undesired events*
++DCL_ARRAY++	p1: name; I	array name	
	p2: type_list; I	element type(s)	
	p3: convar; I	when writeable?	
	p4: array-init; I	initial values	
	p5: indexset; I	array indices	
	p6: data_set; I_OPT	data set name	
			as for ++DCL_ENTITY++ plus:
			%%array too big%%
			%%wrong init value size%%
			%%illegal index set%%

Parameters

There must be as many elements of the array-init as there are elements in the array. Each element must meet the same restrictions as the initial value p4 in ++DCL_ENTITY++.

p2 may only contain or refer to more than one specific type if p3=VAR.

Effects

A one-dimensional array with identifier p1, initial value p4, and index set p5 is declared. Its elements may assume the attributes of any of the types named in p2. If p3=VAR, the elements of the array may be used as !!destination!!s in subsequent operations. The array is declared to belong to data set p6 (or to none if p6 is omitted). Elements of the array can be used wherever an entity of the same specific type could be used. If the spectype of the array has the EXACT_REP attribute and a simple literal given as the initial value of one of its elements is an exact integer multiple of the spectype's resolution, then it will be represented exactly.

2.4. Operand descriptions

2.4.1. Individual parameters

The syntax for !!actual parameter!!s for all EC run-time access programs is shown below. Terms in *italics* are non-terminals. Other terms are terminals, or defined elsewhere in this document. Brackets shown are required.

parameter ::= literal
| entity
(for scalar constants or variables, in which
case the name of the entity is written)
| < *keyword info parameter* >

The UEs %%literal too big%% and %%res too fine%% apply to the first form.

The UEs %%undefined name%% and %uninitialized entity% apply to the second form.

The third form is discussed in detail below.

2.4.1.1. Qualified parameters

Typed literals

keyword ::= LIT
info ::= spectype
parameter ::= literal

The spectype must be previously-declared or built-in, and include the value of the literal in its range. If the spectype has the EXACT_REP attribute and the value of the literal is an exact integer multiple of the spectype's resolution, then the !!typed literal!! will be represented exactly. Otherwise, it will be represented by a value within one-half of the type's resolution. Unless otherwise specified, the use of !!typed literal!!s is optional, but may increase the efficiency of the user's program. See Appendix G.

Applicable UEs: %%undefined name%%, %%wrong type for literal%%

Variable with varying attributes:

keyword ::= ATTR
info ::= spectype

The attributes must be specified by naming a previously-declared specific type. The *parameter* refers to the variable. This form must be used when that variable was declared with more than one spectype in its type_list, but is legal even when the variable was only declared to belong to a single spectype. The specific type must be one that was named, or included in a typeclass that was named, in the type_list of the variable when it was declared. This form of operand is also legal (but not required) for constants, as long as the spectype named is the one that the constant belongs to.

Applicable UEs: %%attribute not given%%, %%illegal sysgen parm%%, %%inappropriate attributes%%, %%undefined name%%, %wrong attributes%

Array elements:

keyword ::= EL
info ::= *parameter* (specifying the index of the array element)

The *parameter* must refer to an array. The index must evaluate to an integer. The element specified is chosen before the operation in which the parameter appears is performed.

Applicable UEs: %illegal array index%, %%illegal sysgen parm%%, %%index not allowed%%

Flooring/rounding/truncating numeric results:

keyword ::= FLOOR | ROUND | TRUNC
info ::=

For this form, the *parameter* must refer to a variable with the EXACT_REP attribute; it may take any form in this section except the FLOOR or ROUND or TRUNC form. Let x be the computed result of the operation in which this operand form appears. If x is an integer multiple of the variable's current !!resolution!! then x is stored into the variable given as the *parameter*. Otherwise, let x_a and x_b be the two integer multiples of the variable's current !!resolution!! closest to x, such that $x_a < x$ and $x_b > x$.

FLOOR has the effect of making x_a the result of the operation. ROUND has the effect of making the result of the operation whichever one of x_a and x_b is closest to x, or either one if they are equidistant. TRUNC has the effect of making the result of the operation whichever one of x_a and x_b has the smallest absolute value.

Applicable UEs: %%illegal round/trunc%%, %%illegal sysgen parm%%

Subrange assertions:

keyword ::= RANGE
info ::= one or more !!interval!!s

Any numeric I or IO !!actual parameter!! may be given using this form; the *parameter* must refer to variable. This form of operand asserts that, at the time of the call, the *parameter* will be within the union of the given !!interval!!(s). If the *parameter* is in turn given using a subrange assertion, that asserts that the both assertions hold simultaneously; i.e., that the value is in the intersection of the assertions' !!interval!!s.

Anyplace a numeric !!destination!! may be used, an operand of this form may be used; the *parameter* must be omitted. This form asserts that the result of the operation will be within the union of the !!interval!!(s). If more than one subrange assertion appears in a !!destination!! list (see next section), it asserts that all of them hold simultaneously; i.e., that the result is in the intersection of the individual assertions' !!interval!!s.

Applicable UEs: %range exceeded%, %%improper subrange assertion%%

Using pointers:

keyword ::= DEREF
info ::=

Anywhere that an entity or array may be used, the reference may be replaced by this form of operand. The *parameter* must refer to a previously-declared entity of the PTR typeclass; it may not take the FLOOR/ROUND/TRUNC or RANGE form. This has the same effect as using the entity or array that is the current value of the pointer.

Applicable UEs: %%illegal ptr target%%, %%illegal sysgen parm%%

Operands that do not change the value of any entity:

keyword ::= NOSTORE
info ::= spectype
parameter ::=

This form may be used as a !!destination!! in a run-time EC access program that computes a real, timeint, or bitstring result of the named spectype. It has the effect of not storing the computed result into any entity. The value can either be used as a !!source!! in a succeeding run-time EC access program as described below, or the computed value can be used to determine the exit of the program (see EC.PGM.1), but not both. This form may not be used more than once in a destination list (see below). To use it as a !!source!! in the succeeding operation, the following conditions must be met:

- the !!command!! using it as a !!source!! immediately follows the !!command!! that used the form as a !!destination!!;
- the spectype given to the !!source!! in the second !!command!! is the same as that given to the !!destination!! in the first !!command!!;
- the first !!command!! has a null !!exit connector!!; and
- the second !!command!! has a null !!name tag!!.

If this form of operand is used as a !!source!! but one or more of the above conditions is not met, a UE will be raised.

Applicable UEs: %%illegal sysgen parm%%, %%inconsistent NOSTORE use%%, %%undefined name%%

2.4.2. Lists of destinations

Any !!actual parameter!! given for an O (output) parameter in an EC access program may be given as a !!list!! of operands. Each element of the !!list!! must be suitable for use as a destination of the operation. All of the parameters will receive the same value (subject to any FLOOR/ROUND/TRUNC effects); the assignments may be made in an arbitrary order or simultaneously.

2.4.3. Lists of operands

Any !!actual parameter!! given in an !!invocation!! of an EC access program may be given as a !!list!! of !!actual parameter!!s. The !!list!!s must all have the same number of elements. Corresponding elements of the !!list!!s must be suitable as individual !!actual parameter!!s for the operation. Such a construct specifies a set of operations, with each set of corresponding !!list!! elements corresponding to one operation. The operations may be done in any order or simultaneously. The set of operations may be interrupted by a parallel process between any two individual operations. An element of a !!list!! given for an O parameter may itself be a !!list!!, as specified in EC.DATA.2.4.2.

Applicable UEs: %%list mismatch%%

2.5. Transfer operations

Program	*Parameters*	*Description*	*Undesired events*
+SET+	p1: see below; I	!!source!!	
	p2: see below; O	!!destination!!	
			%inconsistent lengths%
			%range exceeded%
++SET++	p1: see below; I	!!source!!	
	p2: see below; O	!!destination!!	
			%%inconsistent lengths%%
			%%range exceeded%%

Parameters

p1 and p2 must be both real, or both timeint, or both bitstrings of the same length, or both pointers of the same specific type.

Effects

p2 = the value of p1 before the execution of the program.

2.6. Numeric operations

2.6.1. Numeric comparison operations

Program	*Parameters*	*Description*	*Undesired events*
			None
+EQ+			
+NEQ+			
+GT+			
+GEQ+			
+LT+			
+LEQ+	p1: see below; I	!!source!!	
	p2: see below; I	!!source!!	
	p3: boolean; O	!!destination!!	
	p4: see below; I	!!user threshold!!	
+MAX+			
+MIN+	p1: see below; I	!!source!!	
	p2: see below; I	!!source!!	
	p3: see below; O	!!destination!!	
	p4: see below; I	!!user threshold!!	

Parameters

p1, p2, (p3 for +MAX+ and +MIN+), and p4 must be either all real types or all timeint types.

Effects

+EQ+	p3 = (p1 = p2) †
+NEQ+	p3 = NOT (p1 = p2) †
+GT+	p3 = p1 - p2 is positive and NOT (p1 = p2) †
+GEQ+	p3 = (p1 = p2) † OR (p1 - p2 is positive)
+LT+	p3 = p1 - p2 is negative and NOT (p1 = p2) †
+LEQ+	p3 = (p1 = p2) † OR (p1 - p2 is negative)

+MAX+	p3 = p1 if +GT+(p1,p2, ,p4) returns $true$ and p3 = p2 otherwise.
+MIN+	p3 = p1 if +LT+(p1,p2, ,p4) returns $true$ and p3 = p2 otherwise.

2.6.2. Numeric calculations

Program	*Parameters*	*Description*	*Undesired events*
+ABSV+			
+COMPLE+	p1: see below; I	!!source!!	
	p2: see below; O	!!destination!!	
			%range exceeded%
+ADD+			
+MUL+			
+SUB+			
+SIGN+	p1: see below; I	!!source!!	
	p2: see below; I	!!source!!	
	p3: see below; O	!!destination!!	
+DIV+	p1: see below; I	!!source!!	
	p2: see below; I	!!source!!	
	p3: boolean; I	check for success?	
	p4: see below; O	!!destination!!	
			%range exceeded%
			%divide by zero%
			%%variable parm%%

Parameters

+ADD+
+ABSV+
+COMPLE+
+SIGN+
+SUB+ Either

(1) all operands real, or
(2) all operands timeint

+MUL+ Either

(1) all operands real, or
(2) one of p1 or p2 real, the other operands timeint

+DIV+ Either

(1) p1,p2 and p4 real, or
(2) p1 and p2 timeint and p4 real, or
(3) p1 timeint, p2 real, p4 timeint.

p3 must be given by a literal or ascon.

Effects

+ABSV+	p2 = magnitude(p1)
+ADD+	p3 = p1 + p2
+COMPLE+	p2 = - p1

† Definition of equality: absv(p1 - p2) ≤ !!user threshold!!.

+MUL+ p3 = p1 × p2

+SIGN+ p3 = sign(p1) × absv(p2), where sign(0) is defined to be 0.

+SUB+ p3 = p1 - p2

+DIV+ If p4 is given as <RANGE <LT ub> p4> (or <RANGE <LE ub> p4>) AND absv(ub) ≤ (or <) absv(p1÷p2) then:
(a) if p3=$true$ then sign(p4)=sign(p1÷p2), magnitude of p4 is undefined, and the program referred to by the built-in program pointer DIV_FAIL is invoked;
(b) if p3=$false$ then p4 is undefined.
Otherwise, p4 = p1÷p2.

Built-in Objects

DIV_FAIL A built-in program pointer variable whose attribute is (E1 SCALAR) (defined in EC.PGM.2.3). DIV_FAIL has no initial value; %uninitialized entity% applies.

2.6.3. Operations converting other types to reals

Program	*Parameters*	*Description*	*Undesired events*
			%range exceeded%
+R_BITS_2COMP+ +R_BITS_POSITIVE+ +R_BITS_SIGNMAG+	p1: bitstring; I p2: integer; I p3: real; O	!!source!! !!radix pt ident!! !!destination!!	
+R_BITS_BCD+	p1: bitstring; I p2: integer; O	!!source!! !!destination!!	
+R_TIME_HOUR+ +R_TIME_MIN+ +R_TIME_MS+ +R_TIME_SEC+	p1: timeint; I p2: real; O	!!source!! !!destination!!	

Effects

+R_BITS_2COMP+ p3 = real value equivalent to p1 assuming that bitstring p1 is a two’s complement representation [ADP, page 37] of the number, with the radix point specified by p2.

+R_BITS_BCD+ p2 = integer value equivalent to p1, assuming that bitstring p1 represents a positive number using the "8421" or "direct binary coding" BCD representation scheme [ADP, page 12]. If the length of p1 is not 0 modulo 4, this operation will produce a result as if it were padded on the left with 0:B so that its length is 0 modulo 4.

+R_BITS_POSITIVE+ p3 = real value equivalent to p1 assuming that bitstring p1 is the base 2 representation of a positive number, with bit 0 the most significant bit, and the radix point is specified by p2.

+R_BITS_SIGNMAG+
p3 = real value equivalent to p1 assuming that bitstring p1 is a sign magnitude representation of the number, with bit 1 the most significant bit of the magnitude, bit 0 the sign bit, and the radix point specified by p2.

+R_TIME_HOUR+ p2 = a real value giving the time p1 in hours.

+R_TIME_MIN+ p2 = a real value giving the time p1 in minutes.

+R_TIME_MS+ p2 = a real value giving the time p1 in milliseconds.

+R_TIME_SEC+ p2 = a real value giving the time p1 in seconds.

2.6.4. Operations converting to time intervals

Program	*Parameters*	*Description*	*Undesired events*
+T_REAL_MS+ +T_REAL_SEC+ +T_REAL_MIN+ +T_REAL_HOUR+	p1: real; I p2: timeint; O	!!source!! !!destination!!	%range exceeded%

Effects

+T_REAL_MS+ p2=timeint value equivalent to p1 assuming p1 to specify the time interval in milliseconds.

+T_REAL_SEC+ p2=timeint value equivalent to p1 assuming p1 to specify the time interval in seconds.

+T_REAL_MIN+ p2=timeint value equivalent to p1 assuming p1 to specify the time interval in minutes.

+T_REAL_HOUR+ p2=timeint value equivalent to p1 assuming p1 to specify the time interval in hours.

2.7. Operations for the bitstring type class

Bits in all bitstring types are numbered from 0 upward. We refer to bit 0 as the leftmost bit and a shift of information from higher numbered bits to lower numbered bits as a left shift.

2.7.1. Bitstring comparison operations

Program	*Parameters*	*Description*	*Undesired events*
+EQ+ +NEQ+	p1: bitstring; I p2: bitstring; I p3: boolean; O	!!source!! !!source!! !!destination!!	None

Effects

+EQ+ p3 = (p1 = p2) †

+NEQ+ p3 = NOT (p1 = p2) †

† Definition of equality: length(p1) = length(p2) and for all i: 0 ≤ i < length(p1), bit i of p1 = bit i of p2

2.7.2. Bitstring calculation operations

Program	*Parameters*	*Description*	*Undesired events*
+AND+			
+CAT+			
+MINUS+			
+NAND+			
+OR+			
+XOR+	p1: bitstring; I	!!source!!	
	p2: bitstring; I	!!source!!	
	p3: bitstring; O	!!destination!!	
			%inconsistent lengths%
+NOT+	p1: bitstring; I	!!source!!	
	p2: bitstring; O	!!destination!!	
+SHIFT+	p1: bitstring; I	!!source!!	
	p2: integer; I	shift length	
	p3: bitstring; O	!!destination!!	

+REPLC+	p1: bitstring; I	!!source!!	
	p2: integer; I	source start position	
	p3: integer; I	dest’n start position	
	p4: integer; I	length	
	p5: bitstring; I	background !!source!!	
	p6: bitstring; O	!!destination!!	
			%nonexistent position%
			%inconsistent lengths%

Effects

+AND+	p3 = p1 AND p2
+CAT+	p3 = p1 followed by p2
+MINUS+	p3 = p1 AND (NOT p2)
+NAND+	p3 = NOT (p1 AND p2)
+NOT+	p2 = NOT p1
+OR+	p3 = p1 OR p2
+REPLC+	p6[p3:p3+p4-1] = p1[p2:p2+p4-1] and p6[all other bits] = corresponding bits in p5
+SHIFT+	p3 = shift of p1 by p2 positions to the right (or -p2 positions to the left). The vacated bits are set to "0:B".
+XOR+	p3 = (p1 AND (NOT p2)) OR (p2 AND (NOT p1))

2.7.3. Operations converting to bitstring

Program	*Parameters*	*Description*	*Undesired events*

%left truncation%

+B_REAL_2COMP+		
+B_REAL_POSITIVE+		
+B_REAL_SIGNMAG+	p1: real; I	!!source!!
	p2: integer; I	!!radix pt ident!!
	p3: bitstring; O	!!destination!!
+B_REAL_BCD+	p1: integer; I	!!source!!
	p2: bitstring; O	!!destination!!

Effects

+B_REAL_2COMP+ p3 = two's complement representation of p1, such that the radix point of the resulting bitstring is positioned according to p2. Bit 0 of p3 will be the most significant. The operation truncates all bits beyond the highest numbered bit in the !!destination!! bitstring.

+B_REAL_BCD+ p2 = bitstring representation of ABSV(p1), using the "8421" or "direct binary coding" BCD representation scheme [ADP, page 12].

+B_REAL_POSITIVE+ p3 = bitstring representation of ABSV(p1), such that the radix point of the resulting bitstring is positioned according to p2. Bit 0 of p3 will be the most significant bit. The operation truncates all bits beyond the highest numbered bit in the !!destination!! bitstring.

+B_REAL_SIGNMAG+ p3 = sign magnitude representation of p1, such that the radix point of the resulting bitstring is positioned according to p2. Bit 0 will be the sign bit and bit 1 the most significant bit of the magnitude. The operation truncates all bits beyond the highest numbered bit in the !!destination!! bitstring.

2.8. Operations for the pointer type class

Except for the transfer operations specified in EC.DATA.2.5, there are no operations provided for pointers.

3. Local type definitions

array-init A !!list!! of initial values for an array.

attribute An attribute for a bitstring specifies length, and must be given by a real entity whose value is a positive integer.

A real or timeint attribute is a !!list!!:

(*lower-bound upper-bound !!resolution!!* EXACT_REP)

The fourth element is optional. The lower bound and upper bound are often collectively called *range* (see !!range!!). If a type has the EXACT_REP attribute, then results to be stored into a variable of that type are allowed to be floored/truncated/rounded (at the discretion of the user) to a value that is an integer multiple of the variable's current !!resolution!!. !!range!! and !!resolution!! for reals must given by real entities, and by timeint entities for timeints.

A pointer attribute is either

(*spectype* SCALAR) or (*spectype* ARRAY).

where *spectype* is the name of a previously-declared or built-in specific type. A pointer with the first attribute can refer to scalars of the given spectype (and only that spectype); a pointer with the second attribute can refer to arrays of that spectype (and only that spectype).

Attributes for other typeclasses are given in EC.PGM.2, EC.SMPH, and EC.TIMER.

bitstring An ordered list of values, each value represented by *0* or *1*. The number of such values is called the *length* of the bitstring. A bitstring literal is written as a string of *0*s and *1*s suffixed by *:B*. E.g., *0:B* is a bitstring of length 1 and *1011:B* is a bitstring of length 4. See also "boolean".

boolean Bitstring of length 1. Where convenient, *$true$* may denote *1:B*, *$false$* may denote *0:B*.

convar Either *ASCON* (meaning constant that will not change without reassembly) or *LOADCON* (meaning constant that may be changed by a memory loading device while the program is not running) or *VAR* (meaning variable).

data_set A group (previously declared by ++DCL_DATA_SET++) of user-defined entities that the user may rank according to desired access speed.

data-set-reln A partial ordering on the set of all data sets, given as a !!relation!!.

indexset A set of permissible indices. Only sets of contiguous integers may be created. The set must be specified in the following way:

(si li)

where *si* denotes the smallest index and *li* denotes the largest index. Both si and li must be integer ascons or literals. For example, (7 12) indicates a six-element array indexed by the integers from 7 through 12. (-4 -4) indicates a one-element array whose index is -4.

integer A member of the real typeclass that has the EXACT_REP attribute and whose !!resolution!! = 1.

name An identifier for an object. The syntax is:

```
name ::=      namehead | namehead nametails
namehead ::=  letter | bracketedname
nametails ::= nametail | nametails nametail
nametail ::=  namehead | digit | '_' | '-' | '.'
leter ::=     'A' | ... | 'Z' | 'a' | ... | 'z'
digit ::=     '0' | ... | '9'
bracketedname ::= '$' not('${}') '$'
                  | '/'           not('/{}') '/'
                  | '//'          not('/{}') '//'
                  | '+'           not('+{}') '+'
                  | '++'          not('+{}') '++'
                  | '%'           not('%{}') '%'
                  | '%%'          not('%{}') '%%'
                  | '#'           not('#{}') '#'
```

not('x') ::= any nonempty sequence of ASCII characters in the range octal
040 through 176 inclusive and not containing a character in the string *x*.

pointer A type that provides indirect referencing to other declared entities. A pointer literal is given by <REF x> where *x* is another non-literal entity or array; *x* may not be an i/o data item (see EC.IO).

real An approximation to conventional real numbers. Real literals are denoted in one of the following formats:

standard decimal notation: e.g., -112.345, .000234, 127

exponent notation: decimal number followed by :E*n*, or integer literal followed by :P*n*, where *n* is an integer. The first means the decimal number multiplied by 10 raised to the *n* power; the second means the integer multiplied by 2 raised to the *n* power.
For example: 1.12345:E2 (=112.345), 2.34:E-4 (=.000234), 1:P4 (=16), -10:P-3 (=-1.25).

spectype An identifier that has been previously declared as a type in a ++DCL_TYPE++ operation, or the name of a spectype built in to the EC. The latter includes BOOLEAN (representing the built-in bitstring type *boolean*), as well as those named in EC.PGM.2 and EC.IO.

timeint Representation of a time interval. Literals of type timeint are denoted by using the name of one of the real-to-timeint conversion programs of EC.DATA.2.6.4 and a real literal. The form is:

< TIMEINT program-name real >

where *real* is either a real literal or the name of a real ascon. The value thus specified is that which would be returned by the named program were it called with the real as the input parameter. For example, <TIMEINT +T_REAL_SEC+ 4.0> denotes a timeint value of 4 seconds.

typeclass Either *BITS* (meaning bitstring), *PTR* (meaning pointer), *REAL*, or *TIMEINT* (meaning time interval). Other values are *SEMAPHORE* (see EC.SMPH), *PGM* (see EC.PGM.2), and *TIMER* (see EC.TIMER).

type_list Either a spectype, or the name of a type_list given as p1 of ++DCL_TYPE_CLASS++, or the name of a built-in EC typeclass, or a !!list!! whose elements are these things.

version A version name applicable to the specific type being declared. Version names and characteristics are listed in Appendix F.

4. Dictionary

!!destination!!
An O or IO !!actual parameter!! to an EC access program.

!!interval!! A specification of a numeric interval. The syntax is:

```
!!interval!! ::= < GE  value >
             | < GT  value >
             | < LE  value >
             | < LT  value >
             | < LIM value1 value2 >
             | < LIME value1 value2 >
```

All values must be given by numeric ascons or literals. The LIM form specifies the conjunction of <GT *value1*> and <LT *value2*> if *value1* < *value2* and the disjunction otherwise. The LIME form specifies the conjunction of <GE *value1*> and <LE *value2*> if *value1* ≤ *value2* and the disjunction otherwise.

!!list!! A sequence of zero or more elements enclosed in parentheses. There is a shorthand for specifying sequences of elements:

< RPT *count* !!list!! >

This is equivalent to a !!list!! whose items are the elements of the embedded !!list!! written in sequence *count* times; *count* must be a positive integer ascon or literal.

!!radix pt ident!!
Interpreting the bitstring as a binary real number with bit 0 the most significant bit, 2 raised to the *!!radix pt ident!!* power is the significance of the rightmost (highest numbered) bit. For

instance, a value of zero means that the bitstring represents an integer.

!!range!! The set of values between (and including) the lower bound and upper bound of a numeric data type.

!!relation!! A set of ordered pairs. In EC, a !!relation!! is specified by giving two !!list!!s; the set of ordered pairs is that obtained by taking the cross-product of the !!list!!s. The syntax is:

(!!list!! !!list!!)

!!resolution!! The maximum difference between any two consecutive representatives of the values of a real or timeint data type.

!!source!! An I (input) !!actual parameter!! to an EC access program.

!!typed literal!!

A form of operand in which a spectype is given along with a literal value. See EC.DATA.2.4.1.1.

!!user threshold!!

A difference that user programs specify for a comparison operation; i.e., two numbers whose difference is less than or equal to this are considered equal.

!!version 1 characteristic!!

For numeric types of version R1 or T1, the value of $[\mathrm{MAX}(|lb| + res,|ub + res|) / 2^{\lfloor \log_2 res \rfloor \rceil + 1}$ (where *lb*, *ub*, and *res* are the lower bound, upper bound, and resolution, respectively) of the type.

For bitstrings of version B1, the length.

5. Undesired event dictionary

%%array too big%%

An array was declared to contain more than #max nbr array elements# elements.

%%attribute not given%%

The <ATTR ... > form of operand specification was not used when an entity declared to belong to more than one spectype was used as an operand.

%divide by zero%

A user program attempted to divide by zero.

%illegal array index%

The index supplied in an <EL ... > operand is not in the index set of the array.

%%illegal index set%%

The index set of an array is either not in ascending order or given by integer ascons or literals.

%%illegal length%%

The length of a bitstring type is less than one, or greater than the maximum allowed, as given by a sysgen parameter.

%%illegal ptr target%%

An attempt was made to cause a pointer to point to a literal or to an i/o data item.

%%illegal round/trunc%%

A user used the <FLOOR ...> or <ROUND ... > or <TRUNC ... > form of an operand for a variable that does not have the EXACT_REP attribute or is not a !!destination!!, or used both forms in the same operand specification.

%%illegal sysgen parm%%

The user supplied a parameter to a system-generation-time program in one of the following forms:

<ATTR ... >
<EL ... > when the index was not a constant or literal;
<FLOOR ... >
<ROUND ... >
<TRUNC ... >
<RANGE ... >
<DEREF ... >
<NOSTORE ... >

%%improper subrange assertion%%

Either (a) the <RANGE !!interval!!> form of subrange assertion was used as a !!source!! or IO !!actual parameter!!, or (b) the <RANGE !!interval!! parameter> form of subrange assertion was used as a !!destination!!, or (c) the parameter in a <RANGE !!interval!! parameter> operand was not a numeric variable, or (d) a subrange assertion was given as a destination of an operation that does not compute a numeric result; or (e) subrange assertions were given as the only elements of a !!destination!! list; or (f) the intersection form of subrange assertion was given, and the intersection between at least two of the intervals specified was empty.

%%inappropriate attributes%%

Either (a) in a type declaration, attributes of the wrong typeclass were given; or (b) in a qualified parameter, attributes were specified by naming a spectype not included in the entity's declaration.

%inconsistent lengths%
%%inconsistent lengths%%

The length of the result of a bitstring operation differs from the length of the destination variable, or (for +AND+, +MINUS+, +NAND+, +OR+, or +XOR) the lengths of the two !!source!!s are not the same.

%%inconsistent NOSTORE use%%

A <NOSTORE spectype> operand form either (a) gave a spectype not in the real, bitstring, or timeint typeclass; (b) appeared more than once in a destination list; (c) was used as a !!source!! in a statement not immediately following a statement in which it was used as a !!destination!!, or in a statement with a non-null !!name tag!!; or (d) used as a !!source!! with a spectype not matching that of the previous statement.

%%index not allowed%%

The <EL ...> form of operand specification was used for an entity that is not an array.

%left truncation%

The most-significant bits are lost in a real to bitstring conversion. This results from the user specifying a radix point too close to the most significant bit in the destination bitstring.

%%list mismatch%%

Two !!list!!s which were required to have the same number of elements did not.

%%literal too big%%

The value of a literal is greater in magnitude than that allowed for an entity of that typeclass, as given by a system generation parameter.

%%malformed attributes%%

The attribute for a real or timeint type was given with a lower bound greater than the upper bound.

%%name in use%%

In a declaration or ++REGION++ program call, the user has introduced a name that is already in use as the name of one of the following:

- a built-in object or spectype;
- an EC access program;
- an EC UE;
- an EC reserved word;
- an EC system generation parameter;
- a user-defined spectype, entity, array, data_set, type_class, or region;
- a label in a user-defined program; or
- an entrance or exit in a user-defined program spectype.

%nonexistent position%

A user has specified (1) a start position that does not exist in the bitstring; or (2) a start position and a length that define a substring not contained in the bitstring.

%range exceeded%
%%range exceeded%%

The value being stored into a variable is outside the !!range!! of the variable's spectype; or a variable's value or the result of an operation was not in the !!interval!! specified in a subrange assertion.

%%range too great%%

The magnitude of the declared !!range!! exceeds the maximum allowed for that typeclass, as given by a system generation parameter.

%%res too fine%%

Declared resolution (or implied resolution of a literal) was less than the minimum allowed for that typeclass, as given by a system generation parameter.

%%undefined name%%

A name has been used which is not a builtin name, nor which has been given an explicit meaning by its used in an EC sysgen program invocation, nor an implicit meaning by its use as a label.

%uninitialized entity%

An entity or array element that was declared with initial value UNDEF, and not subsequently given a value, was used as an I or IO operand to an EC access program.

%%unknown initial value%%

A variable has been used as an initial value of a declared entity or array.

%%untyped literal%%

An initial value of an entity or array element belonging to more than one specific type was given by a simple literal.

%%variable parm%%

User supplied a variable or loadcon for an !!actual parameter!! when an ascon or literal was called for.

%%varying constant%%

A user sought to declare a constant as belonging to more than one specific type.

%%version characteristic exceeded%%

The declared type exceeded #max *v* characteristic#, where *v* is the version named in the declaration.

%wrong attributes%

The attributes specified for a varying-attribute variable used as a !!source!! are not the same as for when that variable was most recently used as a !!destination!!.

%%wrong init value size%%

The set of initial values is not the same size as the array.

%%wrong init value type%%

A simple literal used as an initial value is not in the domain of the type of the entity or array element being initialized; or a !!typed literal!! or constant given as an initial value is not of a spectype in common with the entity or array being declared.

%%wrong type for literal%%

In a !!typed literal!!, the given literal was not in the domain of the named spectype.

6. System generation parameters

#max B1 characteristic#

Type: integer. The bitstring types of version B1, the largest allowable value of the !!version 1 characteristic!!.

#max nbr array elements#

Type: integer. The maximum number of elements allowed to be contained in any array.

#max real ascon#

Type: real.

#max timeint ascon#

Type: timeint. Maximum allowable magnitude for a (real, timeint) ascon or literal.

#max real loadcon#

Type: real.

#max timeint loadcon#

Type: timeint. Maximum allowable magnitude for a (real, timeint) loadcon. The value is greater than or equal to that of #max real ascon# or #max timeint ascon#, respectively.

#max R1 characteristic#

Type: real. For real types of version R1, the largest allowable value of the !!version 1 characteristic!!.

#max T1 characteristic#

Type: real. For timeint types of version T1, the largest allowable value of the !!version 1 characteristic!!.

#max real range#

Type: real.

#max timeint range#

Type: timeint. Maximum allowable magnitude for the absv(upper bound - lower bound) for a (real, timeint) type.

#max token length#
Type: integer. The maximum number of characters allowed in a token.

#min real resolution#
Type: real.
#min timeint resolution#
Type: timeint. Minimum allowable resolution for a (real, timeint) entity.

CHAPTER 2

EC.IO: Input/Output

1. Introduction

This module implements two types of bitstring entities known as input data items and output data items, which are used to communicate between the computer and external devices. This interface also includes facilities for i/o used during channel diagnostics. User programs are able to check to see if an external communication has been successful.

Each data item may be enabled or disabled by user programs. When enabled, communication with the outside world is possible. The values of input data items may be set by external devices. The values of output data items are transmitted to external devices. When a data item is disabled, its connection with the outside world is severed.

Each input data item is characterized as either *read-only* or *read-write*. Each output data item is characterized as either *write-only* or *read-write*. The rules for their usage are given in the following table. If the value of an output data item is not initialized before run-time (by use of ++SET++), its value is determined by this module.

Kind of data item		**Enabled, disabled?**	**May be used as:**
input	read-only	disabled	!!source!!
		enabled	!!source!!
	read-write	disabled	!!source!! or !!destination!!
		enabled	!!source!!
output	write-only	disabled	!!destination!!
		enabled	!!destination!!
	read-write	disabled	!!source!! or !!destination!!
		enabled	!!source!! or !!destination!!

Within these constraints, an input or output data item may be used exactly as other bitstring variables.

In addition to the input data items described above, some input from the outside world is handled only through semaphores. For these inputs, which correspond to transient events occurring in external devices, a semaphore is incremented when the event occurs. There are no corresponding bitstrings for these inputs.

2. Interface overview

2.1. Access programs

Program	*Parameters*	*Description*	*Undesired events*

+DISABLE+	p1: dataitem; I	name of data item	%already disabled%
+ENABLE+	p1: dataitem; I	name of data item	%already enabled%

Effects

+ENABLE+ Enables transmission to/from the external environment. If p1 is an input data item, then external values for this input item will now become available internally as soon as practicable. If p1 is an output data item, the value is now available externally. If the item is read-write input, use of the item as a !!destination!! in an EC statement is now prohibited until disabled. At system-generation time, all data items are enabled.

+DISABLE+ Transmission to/from the external environment will be inhibited. If the item is read-write input, it may now be used as a !!destination!! in an EC statement. If the item is read-write output, it may now be used as a !!source!! in an EC statement.

2.2. Access programs for IO diagnostics

Program	*Parameters*	*Description*	*Undesired events*
+TEST_AC+ +TEST_CSA+ +TEST_CSB+ +TEST_DC+ +TEST_DIOW1+ +TEST_DIOW2+ +TEST_DIOW3+ +TEST_XACC+ +TEST_YACC+ +TEST_ZACC+	p1: boolean; O	!+io test result+!	None

Effects

These programs report the results of input/output hardware diagnostic tests. If the test is performed periodically or independent of user request, the result given will be that of the most recent test. If the test is performed on request, the command will initiate the test and report the result when the test is complete. In addition, the following effects are observable.

+TEST_AC+ This program reports the results of the AC signal converter check. It may interfere with output, when the data item is

//BRGDEST// //GNDTRK//
//RNGHND// //RNGTEN// //RNGUNIT//
//STEERAZ// //STEEREL//

+TEST_CSA+ This program reports the results of the cycle-steal channel A and serial channel 1 check. It may interfere with output, when the data item is

//ASAZ//
//ASEL//
//ASLAZ//
//ASLEL//
//ASLCOS//
//ASLSIN//
//AZRING//
//BAROHUD//
//FLTDIRAZ//
//FPMAZ//
//FPMEL//
//HUDCTL//
//LSOLCUAZ//
//LSOLCUEL//
//MAGHDGH//
//MAPOR//
//PTCHANG//
//PUACAZ//
//PUACEL//
//ROLLCOSH//
//ROLLSINH//
//USOLCUAZ//
//USOLCUEL//
//VERTVEL//
//VTVELAC//
//XCOMMF//
//XCOMMC//
//YCOMM//

It may interfere with input, when the data item is /LOCKEDON/ or /SLTRNG/.

+TEST_CSB+ This program reports the results of the cycle steal channel B and serial channel 2 check. It may interfere with output, when the data item is

//CURAZCOS// //CURAZSIN// //CURPOS//

and input, when the data item is

/ANTGOOD/ /DGNDSP/ /DRFTANG/
/DRSFUN/ /DRSMEM/ /DRSREL/
/ELECGOOD/

+TEST_DC+ This program reports the results of the DC signal converter check. It may interfere with output, when the data item is

//FPANGL// //GNDTRVEL// //STERROR//

+TEST_DIOW1+
+TEST_DIOW2+
+TEST_DIOW3+

These programs report the results of the checks on discrete input and output word pairs 1, 2, and 3 respectively. These programs may interfere with output, when the data item is

//DOW1// //DOW2//

and input, when the data item is

/DIW1/ /DIW2/ /DIW3/
/DIW4/ /DIW5/ /DIW6/
/ANTGOOD/ /DGNDSP/ /DRFTANG/
/DRSFUN/ /DRSMEM/ /DRSREL/
/ELECGOOD/ /LOCKEDON/ /SLTRNG/
/SINSDD/

+TEST_XACC+
+TEST_YACC+
+TEST_ZACC+

These programs report the results of checks on the accelerometer and torque registers associated with the X, Y, and Z axes of the IMS respectively. These programs may cause the IMS to lose its alignment and velocities, and may interfere with output, when the data item is

//XGYCOM// //YGYCOM// //ZGYCOM//

and input, when the data item is

/XGYCNT/ /XVEL/ /YGYCNT/
/YVEL/ /ZGYCNT/ /ZVEL/

2.3. Built-in objects

The names of all data items are listed in Appendix E of this document. The following undesired events may occur when data items are used in EC statements:

%read-write violation%
%%read/write-only violation%%

For each data item *x*, there is a built-in boolean variable *x*_SUCCESS (e.g., /AOA/_SUCCESS) which will be $true$ if and only if the last transmission of item *x* was successful, or the first transmission of *x* has not yet occurred. These variables may not be used as !!destination!!s. %%success item as destination%% applies.

2.4. Events signalled by incrementing a semaphore

For some inputs, an event is signalled (by incrementing a semaphore) when a new value of an input data item has been transmitted. The event is of the form

@T(!+*x* ready+!)

where *x* is the name of the data item.

Some inputs correspond to an event occurring in an external device. When such an event occurs, this module will signal a corresponding event of the form

@T(!+*x* occurred+!)

where *x* specifies the event.

These events and their corresponding semaphores and regions are enumerated in Appendix E.

3. Local type definitions

dataitem The name of any input or output data item. The data items are listed in Appendix E of this document. The semantics of the data items are given in Chapter 2 of [REQ].

4. Dictionary

Any term of the form
!+*x* ready+! The named data item is now available for read operations.

Any term of the form

!+*x* occurred+! The named event has just occurred in an external device.

!+io test result+! true iff the i/o hardware passes built-in test.

5. Undesired event dictionary

%already disabled% A user program has tried to disable a data item already disabled.

%already enabled% A user program has tried to enable a data item already enabled.

%%read/write-only violation%%
A read-only (write-only) data item appears as an output !!actual parameter!! (!!source!!).

%read-write violation%
A program call was executed with a read-write input data item as a !!destination!! when that data item was enabled.

%%success item as destination%%
A builtin *x*_SUCCESS variable (*x* some data item) was used as a !!destination!!.

6. System generation parameters

#max i/o time *x*# Type: timeint. (where *x* is replaced by the name of each data item in turn) The maximum time interval that can elapse between the beginning of the access program that reads/writes the named item, and the time it takes for the external transmission to take place.

#nbr fltrec elements# Type: integer. Defined in Appendix E.

#*x* length# Type: integer. (where *x* is replaced by the name of each data item in turn) The number of bits contained in *x*.

CHAPTER 3

EC.MEM: Memory Module

1. Introduction

This module provides a means for testing the memory hardware of the target computer, and reports whether or not the user's program exceeds the amount of memory available.

2. Interface overview

Program	*Parameters*	*Description*	*Undesired events*
+TEST_MEMORY+	p1: boolean; O	!+memory test result+!	None

Effects

Reports the result of the memory diagnostic test. If the test is performed periodically or independent of user request, the result given will be that of the most recent test. If the test is performed on request, the command will initiate the test and report the result when the test is complete.

3. Local type definitions

None.

4. Dictionary

!+memory test result+!   true iff the memory diagnostic test is passed.

5. Undesired event dictionary

%%not enough memory%%
There is not enough memory available in the target computer to store the user's program.

6. System generation parameters

None.

CHAPTER 4

EC.PAR.1: Process Mechanisms

1.1. Introduction

The process mechanism allows the definition of a set of sequential processes that will proceed in parallel and unknown relative speeds. *Demand processes* are activated when specific events occur. *Periodic processes* may be turned on or off, but are re-started at regular intervals when turned on.

1.2. Interface overview

1.2.1. Access program table

Program	*Parameters*	*Description*	*Undesired events*
++D_PROCESS++	p1: timeint; I p2: program; I	!!deadline!! process body	%inconsistent time parms% %missed deadline% %%sysgen process body%%
++P_PROCESS++	p1: timeint; I p2: timeint; I p3: semaphore; I p4: program; I p5: boolean; I	!!deadline!! !!period!! !!starting event!! process body !!on/off!!	As above, plus: %%illegal synch%% %process completed%
+TEST_INTERRUPTS+	p1: boolean; O	!+interrupt test result+!	None

Parameters

++D_PROCESS++
++P_PROCESS++ The program given as the body must be that of a run-time program.

Effects

++D_PROCESS++ establishes a demand process that becomes active after @T(!+power up+!). The body of the process is the run-time program named by p2. The process remains active until it is suspended as a result of a synchronization operation (see EC.PAR.2) or executes the last statement in its body. During the interval when it is active, it will execute before p1 real time has elapsed. A process that is suspended as a result of a synchronization operation may start again. A process that executes its last statement (i.e., exits from the program invoked as its body) will start again only after a system generation.

++P_PROCESS++ establishes a periodic process that becomes active when the semaphore named by !!starting event!! becomes nonnegative. The body of the process is the run-time program named by p4. While the boolean named by p5 is true, a built in semaphore, NEXT_PERIOD, will be incremented at the start of each !!period!! amount of real time. After the start of a !!period!!, the process will complete execution before p1 real time has elapsed. The process must perform [+DOWN+ NEXT_PERIOD] [+PASS+ NEXT_PERIOD].

If p5 is given as a variable, and that variable becomes false while the !!state!! of the process is active, the process will stop when it waits for the start of its next !!period!! (by invoking [+PASS+ NEXT_PERIOD]).

If p2 is given as a variable, and that variable changes value while the !!state!! of the process is active, the process will change its !!period!! within an amount of time equal to the previous value of p2.

Both

If p1 is given as a variable and that variable changes value while the !!state!! of the process is active, the process will change its !!deadline!! within an amount of time equal to its !!period!! (for periodic processes) or by the time the process next resumes execution after suspension due to a synchronization operation (for demand processes).

If two (or more) processes simultaneously execute sequences of statements that read and/or alter the value of some data, the results are unpredictable because the executions may overlap in time. However, EC access programs are considered indivisible. If two EC access programs are executed simultaneously by two processes, the effect will be as if one of the processes executed its access program before the other; the order is not specified. Note that the !!invocation!! of a user-supplied routine is the execution of a single EC access program, but the execution of the body of that routine is a sequence of EC statements.

+TEST_INTERRUPTS+

Reports the results of the interrupt hardware checks. If the test is performed periodically or independent of user request, the result given will be that of the most recent test. If the test is performed on request, the command will initiate the test and report the result when the test is complete. It may interfere with normal operation of timers and input/output commands in unpredictable ways.

1.2.2. Built-in objects

NEXT_PERIOD

a semaphore variable, private to each periodic process, that will be incremented by the EC at the start of each period. Each periodic process only has access to its own NEXT_PERIOD. Semaphores are described in EC.SMPH.

1.3. Local type definitions

None.

1.4. Dictionary

!!deadline!!

The maximum amount of real-time that can be allowed to elapse between the time that a process can proceed and the time that it reaches the next point of suspension.

!+interrupt test result+!

True iff the interrupt hardware passes built-in test.

!!on/off!! The boolean whose value will be used to start/stop the periodic process in whose definition it appears. Its value must be $true$ whenever the periodic process is supposed to proceed. If it is $false$ when the process next reaches its starting point, the process will be suspended until it becomes $true$ again. Of course, the value may only be changed if the boolean was given as a variable.

!!period!! The timeint whose value will be interpreted as the amount of real-time that should elapse between the beginning of one execution of a periodic process and the beginning of the next execution. If !!period!! is given as a variable, changing its value has the result of changing the period of any process for which it was used as the !!period!!.

!!starting event!! The name of a semaphore that, when becoming nonnegative, will cause the periodic process in which it is named to become active.

1.5. Undesired event dictionary

%%illegal synch%% The body of a periodic process either (a) does not contain the statement sequence [+DOWN+ NEXT_PERIOD] [+PASS+ NEXT_PERIOD]; or (b) contains a +PASS+ operation on some semaphore other than NEXT_PERIOD.

%inconsistent time parms% The timing parameters are contradictory; i.e. !!deadline!! exceeds the current value of !!period!!.

%missed deadline% A periodic process has missed its !!deadline!! because too many demand processes have occurred; or a demand or periodic process has missed its !!deadline!! because its !!deadline!! was less than the CPU time required for it to execute.

%process completed% A periodic process has completed execution by taking an exit from the program that is its body.

%%sysgen process body%% The program given as the body of a process is a sysgen-time program.

1.6. System generation parameters

None

CHAPTER 5

EC.PAR.2: Exclusion Regions

2.1. Introduction

This module allows constraints to be placed on the potential concurrency of processes executing regions of code by defining an exclusion relation among them. Region 1 *excludes* region 2 if starting to execute region 2 is forbidden while region 1 is being executed. Mutual exclusion is a special case of this exclusion relation, which is based on [BELP73].

2.2. Interface overview

2.2.1. Access program table

Program	*Parameters*	*Description*	*Undesired events*
++REGION++	p1: name; I	region name	%%name in use%%
++END_REGION++	p1: region; I	region name	%%region across pgms%%
++EXCLUSION++	p1: exclusion-relation; I		%%illegal exclusion%%

Effects

++EXCLUSION++

If the exclusion relation includes (A B) then no process will begin to execute any statement in region B in the time interval that starts when a process begins execution of any statement in region A and ends before that process begins execution of any statement not in region A. The exclusion relation for all regions is composed of the exclusion relation given in each !!invocation!! of this program. User-defined regions may not exclude EC built-in regions.

++REGION++
++END_REGION++

p1 may be used to stand for the section of code that is enclosed between these two statements; p1 may now be used in an ++EXCLUSION++ statement.

If the last action before the region causes a process to wait, then the process is considered to be inside the region when it is allowed to proceed. If the last statement to be executed in the region is a wait operation, then the process is considered to have left the region when it begins to wait. Including regions in a process will prevent the process from waking up, if doing so would result in a violation of an exclusion region.

If an invocation of a user-defined program is in a region, a process executing the invocation does not leave the region during the execution of the program. If the invoked program contains a region, then the process containing the invocation enters the region when the code contained in the region is executed.

2.3. Local type definitions

exclusion-relation A !!relation!! on regions.

region A name (defined in EC.DATA.3) of an exclusion region that previously appeared in a call to ++REGION++, or the name of a built-in EC region that appears in EC.INDEX.

2.4. Dictionary None.

2.5. Undesired event dictionary

%%illegal exclusion%%
The name of an EC built-in exclusion region appeared in the range of the exclusion relation.

%%region across pgms%%
There is an !!invocation!! of ++REGION++ in the body of a program without an !!invocation!! of the corresponding ++END_REGION++ in the body of the same program.

2.6. System generation parameters None.

CHAPTER 6

EC.PGM.1: Program Construction

1.1. Introduction

Using the facilities of this module, a user can construct programs composed of !!invocation!!s of EC built-in access programs and user-defined programs. This is done by naming the entrances and exits of these programs and describing connections between them. Each exit is connected to one !!invocation!!. The resulting structure is called a !!constructed program!!. On completion of its execution, a program selects an exit; the next program executed will be the one connected to that exit.

All EC access programs have one entrance. Many have one exit, but some (see EC.PGM.1.2.1) have as many exits as there are values in the range of the output parameter. When such a program is executed, it chooses the exit that corresponds to the value it has computed.

A !!constructed program!! is a literal of the typeclass PGM. In EC.PGM.2 we describe the declaration and use of entities of that typeclass. In EC.PGM.3 we describe facilities for invoking programs as closed subroutines.

1.2. Interface overview

1.2.1. Entrances and exits of EC access programs

Every EC access program has exactly one entrance.

Every EC access program that has a single output parameter has n exits, where n is the number of values that can be computed for the output parameter. Each exit is denoted by a literal or ascon of the desired value. The exit taken by the program is the one denoted by the literal or ascon that is equal to the value computed by the program, where equality is defined as in EC.DATA.2.6.1 with threshold = 0.

For purposes of comparison, the spectype of the result of the operation is the same as the !!destination!! given with the operation, or (if more than one !!destination!! is given) the *last* member of the destination !!list!!.

All other EC access programs have exactly one exit.

1.2.2. Entrances and exits of !!constructed programs!!

The entrances and exits of a !!constructed program!! are determined by its specific type. See EC.PGM.2.

1.2.3. Connecting !!command!!s within a !!constructed program!!

1.2.3.1. !!Name tag!!s

All !!invocation!!s of run-time programs are preceded by a !!name tag!!; a !!name tag!! is either the empty string, or a !!list!! of names followed by a colon. Each name may be an entrance included in the attributes of the program spectype of the !!constructed program!! in which the !!invocation!! appears; otherwise it is considered to be a label.

These UEs apply to !!name tag!!s:

%%label name in use%%
%%sysgen name tag%%

1.2.3.2. !!Exit connectors!!

All !!invocation!!s of run-time programs are followed by an !!exit connector!!, in one of the following forms.

!!exit connector!! ::= : connection-list |

The second (null) form pairs all exits of the invoked program to the textually next !!command!!.

connection-list ::= a !!list!! whose elements are connections

connection ::= (exit-list port) | port

The first form pairs all exits in the exit-list with the corresponding port. (The exits must be exits of the program whose invocation this !!exit connector!! follows.)

The second form pairs all otherwise unconnected exits with the port. The second form may appear at most once in a connection-list.

exit-list ::= exit | !!list!! of exits and/or !!interval!!s

An !!interval!! specifies a range of values and thus denotes a set of exits; this form is only legal for EC builtin programs that compute a single output numeric that is numeric.

exit ::= name | value

The second form applies only to EC builtin programs that compute a single output parameter. The value may be given by an ascon or literal, and must evaluate at system generation time to a value that the program can return. The first form applies to all other programs, including user-defined programs.

port ::= label | exit

If a label, the label must be that of a label in the !!name tag!! of any !!command!! in the !!constructed program!! in which this !!command!! appears. If an exit, the exit must be an exit in the attributes of the program spectype of the !!constructed program!! in which this !!command!! appears.

After a !!command!! is executed, the next !!command!! to be executed is determined by the port paired with the exit that the !!command!! selected. If the selected exit is paired with a label, the next !!command!! to be executed will be the one in the same !!constructed program!! whose !!name tag!! contains that label. If the selected exit is paired with an exit to the !!constructed program!! itself, then the execution of the !!constructed program!! is complete. These UEs apply to !!exit connectors!!:

%%ambiguous exit connector%%
%%dest unknown%%
%nowhere to go%
%%not an exit%%
%%sysgen exit connector%%

1.2.4. Order-independent operations -- !!par program!!s

The EC provides a way for the user to specify a set of operations for which the order of execution doesn’t matter. Use of this construct may enhance the user program’s efficiency; see Appendix G. The construct is another form of !!constructed program!! and is called a !!par program!!. Its syntax is given in the definition of that term. The !!command!!s will be executed in an order determined by this module. Each !!command!! in the !!par program!! must have a null !!exit connector!!, and a null !!name tag!!. None may use the <NOSTORE ...> form of operand for any parameter.

%%inappropriate par list%% applies to !!par program!!s.

1.3. Local type definitions

invocation-list A !!list!! of !!invocation!!s.

1.4. Dictionary

!!command!! An !!invocation!! of a run-time program preceded by a !!name tag!! and suffixed by an !!exit connector!!.

!!constructed program!!
A sequence of !!command!!s and/or !!invocation!!s of sysgen programs; equivalently, a program literal. The syntax is

<PGM call-list> | <PAR call-list>

The second form is that of a !!par program!!;

call-list ::= call | call call-list

and

call ::= !!command!! | !!invocation!!

!!exit connector!! An association between the exits of an invoked program and labels in or exits to the !!constructed program!! in which the !!invocation!! appears. The syntax and semantics are defined in Section 1.2.3.2.

!!name tag!! Either the empty string, or a !!list!! of labels or entrances; precedes every !!invocation!! of a run-time program. See Section 1.2.3.1.

!!par program!! A !!constructed program!! that uses the PAR keyword, and whose !!command!!s have null !!exit connectors!! and !!name tag!!s, and do not use the <NOSTORE ...> form of operand specification. The EC will choose the order of execution of the !!command!!s. A !!par program!! is a program literal whose specific type is E1.

1.5. Undesired event dictionary

%%ambiguous exit connector%%
An !!exit connector!! paired the same exit with two different ports.

%%dest unknown%%
Either (a) an !!exit connector!! named a port that was neither a label in any !!command!!'s !!name tag!! in the !!constructed program!!, nor an exit of the !!constructed program!!; or (b) a null !!exit connector!! was given in the textually last !!command!! in a !!constructed program!! that was not a !!par program!!.

%%inappropriate par list%%
The invocation-list contained an !!invocation!! preceded by a non-null !!name tag!! or followed by a non-null !!exit connector!!, or contained a <NOSTORE ...> operand.

%%label name in use%%
A name that is not an entrance appears in a !!name tag!!, and that name is already in use as one of the following:

- a built-in object or spectype;
- an EC access program;
- an EC UE;
- an EC reserved word;

- an EC system generation parameter;
- a user-defined spectype, entity, array, data_set, type_class, or region;
- an exit in a user-defined program;
- a label in the same !!constructed program!!; or
- an entrance of some other !!constructed program!!.

%%not an exit%% The !!exit connector!! following an !!invocation!! contained an exit that is not an exit of the program being invoked.

%nowhere to go% A program took an exit that was left unconnected.

%%sysgen exit-connector%%
An !!invocation!! of a sysgen program was followed by an !!exit connector!!.

%%sysgen name tag%%
An !!invocation!! of a sysgen program was preceded by a non-null !!name tag!!.

1.6. System generation parameters None.

CHAPTER 7

EC.PGM.2: Program Entities

2.1. Introduction

This module provides mechanisms for declaring entities of type *program* and assigning a value to them. The EC access programs are built-in program constants; see EC.PGM.2.2.3.2.

2.2. Interface overview

2.2.1. Declaring a program type

To declare specific program types, use the ++DCL_TYPE++ program specified in EC.DATA.2.1, with:

p2 = PGM;
p3 a pgm-attribute, defined in section EC.PGM.2.3;
and the other parameters as described there.

2.2.2. Declaring a program entity

To create an entity of the program typeclass, use ++DCL_ENTITY++ (see Section EC.DATA.2.2.1) with:

p2 = a spectype_list of program spectypes (builtin or user-declared);
p4 = a program literal, or a parameterless program constant whose spectype is named in p2,
or UNDEF;
and the other parameters as described there.

2.2.3. Declaring an array of programs

To create an array of program entities, use ++DCL_ARRAY++ (see Section EC.DATA.2.2.2) with:

p2 as described above; p4 = an array-init (defined in EC.DATA.2.3) of program literals or parameterless program constants whose spectypes are named in p2, or UNDEF; and the other parameters as described there.

2.2.4. Other operations on program entities

Program	*Parameters*	*Description*	*Undesired events*
+SET+			
++SET++	p1: program; I	!!source!!	
	p2: program; O	!!destination!!	
			%%inconsistent pgms%%

Parameters

+SET+	
++SET++	p1 and p2 must be !!list!!s of program entities, with corresponding elements having the same attributes. None may have parameters.

Effects

+SET+ ++SET++	As described in EC.DATA.2.5.

2.2.5. Built-in objects

2.2.5.1. Undesired event programs

Every run-time UE specified in this document is a built-in uninitialized pointer variable with attribute (E1 SCALAR). The program referred to by the pointer will be invoked when the error condition corresponding to the UE definition is detected by the EC program(s) to which the UE applies. It is up to the user to assign a value to each of these pointer variables; %uninitialized entity% applies. Exception: %uninitialized entity% is itself initialized to <REF +S_FAIL_STATE+>.

2.2.5.2. EC Built-in access programs

Each EC access program is a built-in constant of the program typeclass. For most programs, the spectype of the program is not named. If the access program has no parameters, its spectype is E1.

2.2.6. Undesired event detection

UE detection may be turned on and off by the following program.

Program	*Parameters*	*Description*	*Undesired events*
++CHECK++	p1: name; I	run-time UE	
	p2: boolean; I	check for p1?	
			%%not runtime UE%%

Parameters

p1 must be a !!list!! of run-time undesired event names. p2 must be given by an ascon or a literal.

Effects

For each UE named in p1: If p2 = $true$ ($false$) then the UE will be detected (will not be detected) in code subsequent to this !!invocation!! but prior to the textually next !!invocation!! of ++CHECK++ that contains the UE in p1. Initially, the EC will detect no run-time UEs.

2.3. Local type definitions

E1 A built-in specific type of the program typeclass. It is characterized by a single entrance named ENTRANCE1 and a single exit named EXIT1.

pgm-attribute An ordered pair (!!list!! !!list!!). The first !!list!! names the entrance(s) to programs of the type; the second !!list!! names the exit(s) of programs of the type. See EC.PGM.3.3 for a definition of "entrance". %%name in use%% applies to the entrances and exits.

program An entity of the program typeclass previously declared via ++DCL_ENTITY++, or a member of a program array previously declared via ++DCL_ARRAY++, or a built-in EC access program, or a program literal. A program literal is a !!constructed program!! as defined in section EC.PGM.1.

2.4. Dictionary

None.

2.5. Undesired event dictionary

%%inconsistent pgms%%
An assignment was attempted between entities of different attributes; or, one of the entities is not a parameterless program.

%%not runtime UE%%
A name was given that is not the name of a run-time undesired event in the EC.

2.6. System generation parameters

None.

CHAPTER 8

EC.PGM.3: Program Invocation

3.1. Introduction

This module provides mechanisms for invoking programs (either built-in or user-defined) and, in the former case, passing parameters to those programs.

3.2. Interface overview

The syntax for invoking a program is as follows:

!!invocation!! ::= [*pgm parm-list*]

For programs with no parameters, the *parm-list* is empty.

Effects

If an EC run-time access program is named, the effect is that which is specified for that program.

If an EC system-generation-time access program is named, there is no run-time effect; system generation programs are executed before run-time in the order of their occurrence in the source code and before any code is compiled. The values used in compiling the code are the last values computed by this pre-run-time execution.

If a user-defined program is named, the effect is that of executing the !!command!!s in the !!constructed program!! that has been assigned to the program (either as the initial value as described in EC.PGM.2.2.1, or subsequently as described in EC.PGM.2.2.3) beginning at the entrance named.

The following undesired events apply to program !!invocation!!:

%constant destination%
%%constant destination%%
%%entrance incorrectly omitted%%
%%not an entrance%%
%recursive call%
%%undefined name%%
%%wrong num parms%%
%%wrong type%%
%wrong type%

3.3. Local type definitions

entrance — The name of an entrance to the program that is the value of the invoked entity, previously specified by ++ENTRANCE++ for that program.

parm-list

::=
| parm parm-list

parm — An !!actual parameter!! to the program.

pgm

::= program | (program entrance)

The first form may be used when the program only has one entrance.

3.4. Dictionary

!!actual parameter!!

An entity that appears in the parameter list of a program !!invocation!!. The forms that this may take are specified in EC.DATA.2.4.

!!invocation!! A program call; the syntax is defined in Section 3.2 of this chapter.

3.5. Undesired event dictionary

%constant destination%
%%constant destination%%

The user has supplied a constant or a literal as a !!destination!!.

%%entrance incorrectly omitted%%

The user has failed to specify an entrance in an !!invocation!! of a program that has more than one entrance.

%%not an entrance%%

The entrance named in the !!invocation!! is not an entrance to the program being invoked.

%recursive call%

A program has invoked itself, either directly, or indirectly. More formally, for the purposes of this definiition let I be a relation called *invokes* whose domain and range are the set of user-defined EC programs. The ordered pair (*a*,*b*) is in the relation if *a* invokes *b*. The relation is transitive. Then this UE is raised if there exists some program *x* such that (*x*,*x*) is in I.

%%wrong num parms%%

The programmer supplied a different number of !!actual parameters!! than the number called for by the program's specification.

%%wrong type%%

The type of an !!actual parameter!! is not of the type called for in the specification of the program, as indicated in the "Parameters" column of the access program table.

%wrong type%

The !!actual parameter!! failed to meet the type or other constraint specified in the "Parameters" section immediately following the access program table; or, the !!actual parameter!! was given using the <DEREF *ptr*> form of operand description and the pointer was pointing to an entity not of the type called for in the "Parameters" column of the access program table.

3.6. System generation parameters

None.

CHAPTER 9

EC.SMPH: Synchronization Variables and Operations

1. Introduction

This module provides a run-time synchronization mechanism, semaphores, with associated operations. They can be used where exclusion regions cannot express the constraints. This mechanism is based on [BELP73]; the semaphore operations are a more primitive version of Dijkstra's P and V [DIJK68].

Semaphores can also be affected by timers; see EC.TIMER.

2. Interface overview

2.1. Declaring semaphore types

To create specific semaphore types, use the ++DCL_TYPE++ program specified in EC.DATA.2, with:

p2 = SEMAPHORE;
p3 a semaphore-attribute, defined in EC.SMPH.3;
and the other parameters as described there.

2.2. Declaring semaphore entities

Semaphore entities must be declared before they can be used. Use the ++DCL_ENTITY++ program of Section EC.DATA.2.2, with:

p4 (the initial value) given as an integer literal, or UNDEF;
and the other parameters as described there.

2.3. Declaring semaphore arrays

To create an array of semaphores, use the ++DCL_ARRAY++ program of EC.DATA.2.2.2, with:

p4 as described there, with initial values given as integer literals, or UNDEF;
and the other parameters as described there.

2.4. Access programs

Program	*Parameters*	*Description*	*Undesired events*
+DOWN+			%range exceeded% %%illegal up/down%%
+UP+	p1: semaphore; IO		
+SET+	p1: semaphore; I p2: semaphore; O		
+PASS+	p1: semaphore; I		None.

Effects

In this section, we characterize informally the effects of the synchronization operations. For a more precise description, see the formal specifications in [TRACE].

OPERATION	EFFECT ON NAMED SEMAPHORE	EFFECT ON PROCESS STATE(S)
+UP+	incremented by 1	if semaphore ≥ 0 then !!state(waiters)!! := active
+PASS+	none	if the semaphore < 0 then !!state(self)!! := waiting
+DOWN+	decremented by 1	none
+SET+	p2 set to value of p1	same as if the value of p2 were arrived at by the smallest possible number of consecutive operations of +UP+ and +DOWN+.

3. Local type definitions

semaphore — A run-time synchronization object created previously by a user program by calling ++DCL_ENTITY++; or one of the EC's built-in semaphores listed in EC.INDEX.

semaphore-attribute — An ordered pair of integers specifying the lower bound and upper bound of the type.

4. Dictionary

!!state!! — Either *active* or *suspended*; an active process is eligible to proceed; a suspended one is not.

!!state(self)!! — the !!state!! of the process executing the synchronization operation

!!state(waiters)!! — the !!state!! of all processes in the middle of a +PASS+ operation for a particular semaphore

5. Undesired event dictionary

%%illegal up-down%%

A user has attempted an +UP+ or +DOWN+ operation on an EC built-in semaphore, or attempted to assign a value to one.

6. System generation parameters

#max semaphore ascon#

Type: semaphore. Maximum allowable magnitude for a semaphore ascon or literal.

#max semaphore loadcon#

Type: semaphore. Maximum allowable magnitude for a semaphore loadcon. The value is greater than or equal to that of #max semaphore ascon#.

#max semaphore range#

Type: semaphore. Maximum allowable value for absv(upper bound - lower bound) for a semaphore type.

CHAPTER 10

EC.STATE: Extended Computer State

1. Introduction

This module controls and reports transitions between Extended Computer states.

2. Interface overview

2.1. Access programs

Program	*Parameters*	*Description*	*Undesired events*
+S_FAIL_STATE+			None

2.2. Events signalled by incrementing a semaphore

Event	**Semaphore**	**Region**
@T(!+power up+!)	ECPOWUP	ECPOWUP_reg
@T(!+failed state+!)	ECFAILED	ECFAILED_reg

Effects

+S_FAIL_STATE+

!+failed state+! = $true$, ECFAILED is incremented, and an internal shutdown procedure is executed.

@T(!+failed state+!)

Programmers should assume that when #close down time# has elapsed after this event, no more software actions can occur.

@T(!+power up+!)

The Extended Computer has entered the operating state and is functioning correctly. All demand processes are started.

3. Local type definitions

None.

4. Dictionary

!+power up+! computer is in the operating state and may be assumed to be functioning properly.

!+failed state+! no more software actions may occur more than #close down time# time after !+failed state+! becomes true.

5. Undesired event dictionary None.

6. System generation parameters

#close down time#

Type: timeint. The minimum expected time interval between the moment that the Extended Computer enters failed state and the moment when no more software actions may occur.

CHAPTER 11

EC.TIMER: Timer Facilities

1. Introduction

This module provides facilities for measuring real time intervals via *timers*. A timer is just like a timeint variable except that, when running, it will increment or decrement at a rate commensurate with real time.

A timer may be used anywhere a timeint variable may be used. In addition, there are two additional operations, START_TIMER and HALT_TIMER, that may be used. START_TIMER increments or decrements the timer until a limit is reached.

When a timer is declared, the user may choose between timers that increment and timers that decrement, as well as between timers that halt when they reach their limit and timers that "wrap around". The user may also specify a semaphore that will be incremented when the timer reaches its limit.

2. Interface overview

2.1. Declaring timer types

Timers are a numeric type class, as described in EC.DATA.1. To declare specific timer types, use the ++DCL_TYPE++ program specified in EC.DATA.2.1, with:

p2 = TIMER;
p3 a timer-attribute, defined in section EC.TIMER.3;
and the other parameters as described there.

2.2. Declaring timer entities

Timer entities must be declared before they can be used. Use the ++DCL_ENTITY++ program of Section EC.DATA.2.2.1, with:

p3 = VAR;
p4 the initial value given as a timeint literal, or UNDEF;
and the other parameters as described there.

2.3. Declaring timer arrays

To declare an array of timers, use the ++DCL_ARRAY++ program of EC.DATA.2.2.2, with:

p3 = VAR;
p4 as described there, using timeint literals or UNDEF as initial values;
and the other parameters as described there.

2.4. Access programs

Program	*Parameters*	*Description*	*Undesired events*
++TIMER_EVENTS++	p1: timer; I	timer name	
	p2: semaphore; I	limit-value event	
			None
+START_TIMER+			
+HALT_TIMER+	p1: timer; I	timer name	

2.5. Timer tests

Program	*Parameters*	*Description*	*Undesired events*
+TEST_TIMER+	p1: boolean; O	!+timer test result+!	None

Effects

+HALT_TIMER+ Causes running timer p1 to halt. Halting a non-running timer has no effect.

+START_TIMER+ Causes the value of p1 to be changed in value in real time. The value will be increased or decreased according to the declaration of the specific type to which p1 belongs. According to the declaration of the specific timer type to which p1 belongs, the timer will either stop when it reaches its minimum (maximum) value, or "wraparound"; i.e., continue from its maximum (minimum) value. Starting a running timer has no effect.

++TIMER_EVENTS++ Causes an event to be signalled (by incrementing p2) every time p1 reaches its minimum range value (if p1 is a decrementing timer) or its maximum range value (if p1 is an incrementing timer). The semaphore will be decremented after it is incremented, and both operations will occur inside a region called *p1*_reg where *p1* is replaced by the name of the timer.

+TEST_TIMER+ Reports the results of the timer hardware tests. If the test is performed periodically or independent of user request, the result given will be that of the most recent test. If the test is performed on request, the command will initiate the test and report the result when the test is complete. It may interfere with normal operation of timers and input/output commands in unpredictable ways.

3. Local type definitions

timer The name of a time-keeping mechanism declared previously by a user program.

timer-attribute An ordered 5-tuple of the form

(timeint timeint timeint HALT/WRAP UP/DOWN)

The first three elements specify the lower bound, upper bound, and minimum !!resolution!!, respectively, of entities of the type. The fourth element is either "HALT" (meaning that the timer should stop when it reaches a limit) or "WRAP" (meaning that the timer should wrap around when it reaches a limit). The fifth element is either "UP" (meaning that the timer increments when started) or "DOWN" (meaning that the timer decrements when started).

4. Dictionary

!+timer test result+! true iff the timer hardware passes built-in test.

5. Undesired event dictionary

None.

6. System generation parameters

#max static timer error# Type: timeint. Maximum error associated with reading a timer, independent of the time interval measured; positive.

#max timer error rate# Type: real. Maximum time-dependent error rate of all timers, given as a fraction of the time interval measured.

#max timer range# Type: timeint. Maximum allowable value of absv(upper bound - lower bound) for a timer type.

#max C1 characteristic# Type: real. For timers of version C1, the maximum allowable !!version 1 characteristic!!.

#min timer resolution# Type: timeint. For all timers, the minimum resolution.

APPENDIX A

Design Issues

0. GENERAL

(1) It is arguable whether the Efficiency Guide should be included in this document. One author (Parnas) felt that it confused machine-dependent assumptions with a machine-independent (abstract) interface. However, others felt that making this distinction was not worth the overhead of maintaining a separate document. We have tried to make it very clear that should the implementation change, the Efficiency Guide would change *although user programs would continue to be correct.*

1. EC.DATA

(1) We decided to give the programmer some control over the register, so that he could take care of reducing register loads and stores by being careful with the order of operations. The alternatives we considered were notations much closer to high-level programming languages. These notations make complex expressions easier to read, but require a more sophisticated translator if we are to make efficient use of registers.

(2) There is a danger with fixed point division that the results will be meaningless; this problem occurs when the numerator has more significance than the denominator. An assembly language programmer has some information that he uses to avoid this danger. The only way we can get this information is to ask the programmer to provide it, since it is dependent on the context and meaning of the division.

(3) Two ways were proposed for user programs to indicate the radix of the number for a bitstring-real conversion:

a. by giving an integer literal *i* such that the rightmost bit of the bitstring represents 2 raised to the *i-th* power;

b. by giving an integer literal *i* such that *i* is the number of the bit immediately to the right of the radix pt.

Alternative b most closely resembles the scaling notation used in the current program, but we chose alternative a because most designers felt that it was easier for newcomers to understand and remember.

(4) There are two main reasons for including variables whose attributes may vary: (a) they can be reused at different points in a computation, thereby reducing the amount of space that must be reserved; and (b) they allow the same code to be used to manipulate values in widely differing ranges.

(5) We require the programmer to specify a type for results stored into and retrieved from variables whose attributes can vary. We considered permitting, but not requiring, specification of the type of intermediate results and letting the Extended Computer determine the specific type when the programmer omitted the specification. We ruled out this alternative because it requires a run-time support package to keep track of the specific types of varying-type variables.

(6) We considered several alternatives for providing registers:

a. Having a common register for all type classes. This register can be very simply mapped to the accumulator.

b. Having a separate register for each type class, implementing them with the single accumulator, and leaving the problem of interference between them up to the programmer. This was originally accepted because it is the simplest alternative that provides type checking for results in the register. However it gives away the underlying limitation, and imposes restrictions on the programmer that would not be needed if the underlying hardware had more registers or if there was multi-processor hardware.

c. Having a separate register for each type class, implementing them with the single accumulator, and completely automating the problem of interference between the registers, freeing the programmer from any concern about it. This could be done by saving and restoring the accumulator contents whenever a different register is used. While it would be the most convenient alternative, the overhead would be prohibitive.

d. Having a separate register for each type class, implementing them with the single accumulator, and partially automating the problem of interference between the registers. The programmer would have to indicate when he wants to reuse results in a particular register and when he does not care.

We chose alternative (a) because it is the simplest and treats a register as a variable with varying attributes.

(7) We felt it important that the EC implementation avoid saving contents of a register if they would never be needed and therefore put that burden on the programmer rather than try to do register usage analysis. We considered several ways to allow the programmer to specify whether or not the value in the register would be needed again. Among them:

a. Associate the information with the name of the register.

b. Associate the information with the name of the operation.

We chose (b) because we did not want to have two names for the same object. Further, it allows us to localize the information in a place related to the operations (of which it is a property) rather than the registers.

(8) An earlier version of this interface included operations such as squareroot, exponentiation, log, and root-sum-squared. We decided to move these operations to another module because they can be implemented in a machine-independent fashion. These concerns do not belong in the Extended Computer.

(9) An earlier version of this module had two bitstring sizes, corresponding to halfwords and fullwords on the target computer. We then decided to have only one size because it results in a simpler data type. We finally decided to have bitstrings of any size because we noted that insisting on a fixed but unknown size made it difficult to write efficient but machine independent code. The present choice makes the interface unbiased with respect to word length and puts the burden for effective use of the actual hardware on the implementor of the EC.

(10) We considered specifying bitstring sub-ranges in terms of (startingoint, length) instead of (starting point, ending point). One parameter fewer would be needed on bitstring compares and transfers, and we could avoid the unmatching lengths undesired event. However, we found that people working with bitstrings find it easier to work by identifying the boundary bits.

(11) We considered having the EC monitor arithmetic operations for excessive loss of significance but decided that this was a programmer responsibility and could be done in a machine independent way. This eliminated the undesired event %too much lost significance%.

(12) We considered relegating time to the application data type module and implementing it in terms of reals. We chose to include it in the EC because the concept of time is basic to the specification and implementation of real-time processes in the EC and because the representation should be that used in the hardware timers.

(13) We considered allowing array declarations to be shared by several variables. We found this not particularly useful unless one has operations that take whole arrays as operands.

(14) We decided not to allow array elements to be structures. We lose the ability to have arrays of arrays, but if this were necessary, it could be implemented in a machine independent way and could be provided by some other module.

(15) We considered allowing index sets to be more general, but this seemed unnecessary even for future extensions. Such extensions could be done using the present arrays and the extension would be machine independent. We also considered restricting the lower array bound to be either 0 or 1. This seemed unnecessarily restrictive, especially as it may be desirable to select array indices at sysgen time.

(16) We considered fixing the value of the array index set at declaration time, system generation time, or run time. Declaration time is too restrictive; it is sometimes useful for the array index set to be a system generation parameter. Run time fixing requires dynamic storage allocation, which is not needed or practical

for avionics applications.

(17) We rejected the option of operations that apply to arrays as units, e.g. multiplying arrays by scalars or arrays by arrays. Such operations depend on mathematical algorithms, rather than on characteristics of the computer and can be implemented in a machine-independent way. The present design is the simplest way to hide the hardware addressing mechanism. Extentions can be provided by user programs.

(18) We considered not allowing arrays of variables whose attributes vary at run-time as it might simplify the implementation if all elements had the same attributes at all times. Although the implementation of arrays with varying attributes will probably be less efficient than arrays of fixed attribute elements, this feature is occasionally needed.

(19) More than one reviewer asked if the Extended Computer shouldn't provide stacks as a builtin data structure. If we need stacks, they can be provided using the current EC facilities. The interface to those facilities (probably in the Application Data Types module) would be carefully modelled after the EC. Should we transfer to stack machine, we could move the interface into the EC, and user programs would not have to change. This rationale also applies to floating point arithmetic, multi-dimensional arrays, array operators, etc.

(20) Entity names are global in the EC. This is because that is what avionics computers provide; one can limit the scope of a name (if desired) in a machine-independent way (e.g., using naming conventions, or a pre-processor).

(21) We recognize the need to represent data most efficiently for the operations in which it will be used. Since only users can determine how a datum will be used, the best the EC can do is provide a menu of representations and tell the users what each one is best and worst at doing. Hence, the "version" attribute in specific types.

(22) We considered allowing non-homogenous arrays; that is, arrays with different specific types. However, that would mean that if an operand was a member of such an array, we couldn't discover its type until run-time. Because of this great run-time penalty, we deleted the capability for arrays whose attributes can vary; however, we retain the capability of having arrays whose element attributes can vary.

(23) The +SHIFT+ program used to take a list of bitstrings as its input parameter; it shifted the concatenation of the list. The idea was to allow the same kind of shifting that occurs in the TC-2 between adjacent registers. However, it became clear that the implementation of such a feature would simply +CAT+ the strings together first anyway, and shifting the result, and we would gain no efficiency. So for consistency, +SHIFT+ now takes a single source parameter.

(24) We have made the decision to design the EC to be implemented on a machine that supports fixed-point arithmetic and not floating-point arithmetic. The resolution required in a numeric variable is expressed in our machine as a constant, independent of the magnitude of the value of the variable. In fixed-point represenatations, there is a uniform distance between values that can be represented exactly. In floating-point, the distance is small for small values, large for large values.

Were we to go to a floating-point machine, we would need to enhance the interface, because fixed-distance representations on a floating- point machine would be very inefficient. We would let the user specify a worst-case distance between representatives as a fraction of the value of the variable. The implementation would choose a representation so that the mantissa of the number had a resolution (in the current sense) less than the fraction.

The present interface may be considered a subset of a more complete interface in which we let the user specify resolution either in absolute form (in which case a fixed-point representation would probably be chosen) or in relative form (in which case we would represent the number using the floating-point hardware, or by simulating floating-point). The present interface reflects our decision to make EC apply to typical avionics machines, which are fixed-point.

(25) The pointer typeclass is a recent addition, included when we realized that we had denied users the capability found in all von Neumann machines of indirect referencing, or postponing operand specification until run-time. Since the goal of EC was to abstract from the idiosyncrasies of particular avionics computers, yet provide the capabilities that they provide, this was clearly an appropriate addition to the EC architecture.

(26) When declaring a specific type, it used to be an error to specify a version that did not exist for that particular type. We now say merely that the EC will pick one of its own choosing in that case. That is so that should an Application Data Types (ADT) type ever migrate into the EC because of a change that enhances the type repertoire of the hardware, we would like for user programs to remain correct. However, the versions that we would provide in the EC for that type might be entirely different than what were provided for it in the ADT. The ADT types are documented in [ADT].

(27) The EC has variables whose attributes can vary at run-time, but for a long time we restricted the variation to within a single typeclass; a bitstring variable could only take on bitstring attributes, etc. However, we realized that this restriction prevented programmers from re-using storage in some cases, and in fact was a restriction not present on the actual machine. It turned out to be easy to relax, and so we have done so.

(28) We specify that the maximum value of a loadcon must be at least as big as that of an ascon, because sometimes an ascon must be stored at run-time (e.g., when it is pointed to). This essentially converts it into a loadcon, unbeknownst to the user. We must assure that no ascon is too be to be so stored.

(29) (Obviated by later design) We provide -SAVE operations because on some machines it is possible to do memory-to-memory operations without the use of an accumulator. On others, one must load the accumulator and then store into the destination. We want to hide this difference. If we do it by pretending that we can always do a move without using REG, the implementation will always save and restore REG to simulate that. This may not be needed if the user has no more need for the value in REG. On the other, if we hide it by saying that REG is undefined, the user will write programs that store things unnecessarily when the value is not disturbed. The solution to this dilemna is to allow the user to tell us whether he needs the value in REG again. If so, he asks for a save. If not, the implementation is free to destroy it. In this way, on machines where we need the register, we will not save and restore unnecessarily. Further, the user will never have to save and restore himself, so he will not do it.

(30) We formerly did not allow the EC user to use the <ATTR spectype variable> operand form when the variable only belonged to a single spectype. However, we removed this restriction because (a) doing so hurts nothing; and (b) it facilitates certain possible changes to the code: if the user re-declares his variable so that it **can** change type, he need not change all the places where it was used previously.

(31) Similarly, we made it legal to use the <ATTR spectype entity> form for constants. It hurts nothing (as long as the spectype matches the single declared spectype of the constant), and would allow the declaration of a variable to be changed to a constant without having to change every place it was used as an operand. This kind of change might be useful when developing subsets in which a particular variable never happened to be used as a !!destination!!.

(32) We allow a <NOSTORE ...> operand in a destination list because (a) the user may want to store the result of an operation into a variable yet still set up the accumulator as a !!source!! for the next operation and this will prevent an unnecessary load-register instruction; (b) the user may want to use the <NOSTORE ...> operand as the last element in his destination list, thus setting the spectype of the result for branching comparisons. This may pay off in case the user has a set of exit literals/ascons of a particular spectype already declared; this prevents him from having to invent a new set just to match one of his entity destinations.

(33) We used to have UEs %%literal or ascon too big%% and %%loadcon too big%%. We replaced those two with %%literal too big%%, because (a) the only way you can exceed the max ascon size is with a literal that’s too big; and (b) the maximum value of a loadcon is guaranteed to be greater than or equal to that of an ascon/literal anyway. So just the one UE that applies to literals is needed.

(34) We allow a staggering variety of subrange assertions to be given by the user: unions of intersections of intervals, intersections of unions of intervals, etc., with about a half dozen kinds of intervals available. This might seem to run counter to our philosophy of minimizing the facilities that we provide; however, this generalization costs nothing because we are free to ignore the assertions. Further, we cannot be sure what kind of subranges we might find useful on other target machines. In fact, our implementation will only make use of a small subset of the possible kinds of assertions.

(35) It is an unimplemented feature to declare EXACT_REP time intervals whose resolution is not an exact power of two milliseconds. This was controversial, because the decision is actually a result of how we chose to implement time intervals. We could have chosen differently. Thus, we have the situation where an implementation decision led to change to the interface. The alternative would be to represent all

EXACT_REP time intervals -- but to represent the others inefficiently. If we had done that, our efficiency guide would have stated this fact. We believed that such a statement would have led users to eschew the others, and so we would not have had to implement them anyway. We took a shortcut, and simply chose *a priori* not to implement them. This remains a controversial decision, made in the most part to save project time.

(36) We considered adding facilities for computing arithmetic values at system-generation time. This would be useful when the value of certain entities needs to be a function of the value of previously-declared system generation constants, and we need to be able to express that in declarations. Examples might include the length of a bitstring used to represent the bit version of a real number, whose range and resolution are given by sysgen parms. Another example is where we say that the range of a display is symmetric about zero and we give the upper bound as a sysgen parm; the lower bound must be declared to be the negative of the upper bound. Another example is the range of a spectype, which we might want to make 25% bigger than some target value.

We could get around the need by defining enough new sysgen parms and mandating that their values be functions of other sysgen parms. We feared, however, (a) a big proliferation of names; (b) possibly loops and circular definitions, which we are ill-equipped to detect; and (c) a configuration control problem.

The easiest way to add this facility would be to make "++" versions of all the EC.DATA programs (++ADD++, etc.) but unless we added sysgen-time control flow and variables that expired after sysgen-time, this would be insufficient to solve the examples above. We really need a syntax for expressions that can be used in-place in declarations; [APC] will most likely be the model.

We haven’t added the feature because the need for it isn’t as strong as we expected, and because it isn’t at all clear that it belongs in this module.

(37) We used to call #max R1 characteristic# by the name of #max real ranres ratio#, and it purported to be the single limiting value by which all declared reals would be compared for legality. We realized somewhat late that our definition of #max ranres ratio# was highly implementation-dependent. It gives away the fact that we are using a binary machine with a specific type of arithmetic. In other words, it is specific to the *version* of reals that we are implementing. Another version’s limits might be based on other considerations entirely.

We realized that we needed a separate limiting parameter for each version that we provide, because each version will produce its own limits, expressed in different ways. Should we acquire, say, a second real version, then we will have to introduce a new sysgen parm for it. Thus, we changed the name of the single sysgen parameter we have now (and the UE for exceeding it) to make it clearer that the limits depend on the version, and that new versions will yield new sysgen parameters.

(38) The FLOOR operand form is a recent addition, added when we realized that TRUNC is prejudiced towards a sign-magnitude representation (in which it can be implemented by simply shifing out low-order bits). In a two’s-complement representation, the FLOOR form can be implemented in this way, but TRUNC is more expensive. Further, both are useful operations, one which our users may well need, and both appropriately implemented by this module.

We considered adding two more forms (CEILING, with the obvious semantics, and a form of truncation that chooses the one of two closest representatives with the larger absolute value) but they do not appear to be useful. We can add them later if necessary.

(39) We expect to implement EXACT_REP numbers by right-justifying them within TC-2 words. This means that we always store numbers as exact integer multiples of their resolution; further, it means that under some circumstances we must perform a ROUND operation on a result before storing it in order to meet the EC specifications (because simple truncation would lose too much significance). Thus, under some circumstances, using a simple form of operand is actually more expsensive than, say, doing a FLOOR. Although this is somewhat conterintuitive, we don’t expect many such cases. If the user doesn’t care about the exact representation of such numbers, he wouldn’t have asked for EXACT_REP anyway. However, should this not be the case, we have an option of an alternate storage scheme: storing the numbers right-justified, but keeping one extra bit of significance. Then TRUNC and FLOOR are somewhat more expensive, but no operation is very cheap. This would represent an additional *version* of real numbers.

(40) The Application Data Types module has programs that convert an angle to/from a (degrees,minutes,seconds) triplet, in addition to programs that convert to/from degrees, to/from minutes, and to/from seconds. For consistency, the EC should treat time intervals the same way. However, because there is no demand for such programs, we have decided not to add them (we would make them unimplemented anyway). Should we need them, they would be straightforward additions. Each program would take four parameters: hours, minutes, seconds, milliseconds, and the corresponding timeint.

(41) We used to have a UE called %assertion violation% for giving a subrange assertion that turned out to be false. However, the check for this was identical to that of %range exceeded%, and we would have had to produce almost-identical machine code to perform it, so we deleted it.

(42) We added the BCD real/bitstring conversion programs because there are a couple of input data items that BCD scheme to represent numbers. We thought about having the Device Interface Module do the necessary conversions itself, but it turns out that BCD is a common representation scheme in machines that perform decimal arithmetic, and we are explicitly hiding the numeric base of our target computer.

(43) We added the +MAX+ and +MIN+ programs after a discussion that led us to believe they should available in the ADT module. The reasons were that they seemed to be natural operations to perform on a data type, and that they could be performed more efficiently knowing "inside" information about the entities, such as representation. One example is when +MIN+(x,y) *always* returns x because y's spectype contains no value less than any value in x's spectype. We realized that the same reasoning applied to EC.

(44) We used to have a power-of-two notation for real literals: 2**n for some integer n. A negative number was specified as -2**n. This worked well until we realized that our pseudo-code, which allows expressions, evaluates -2**4 as +16, whereas in EC it is -16. Since the syntax one chooses for literals is not particularly important from a machine abstraction point of view, we decided to take what for the project was the easiest way out and change the EC syntax to eliminate that form altogether, combining it with the exponential notation with which it is similar anyway.

(45) After writing a significant amount of code using the EC interface, we notice a great many specific type declarations that exist merely to allow the declaration of a single entity. Were we designing EC over, we might wish to provide a syntax to specify typeclass, attributes, and version in the entity declaration itself, to avoid the somewhat cumbersome spectype declaration. We considered this early in the design phase, but rejected it to encourage users to share spectype declarations among several entities. Sometimes that has worked (so we should keep the spectype declaration statement to provide a shorthand name for a <typeclass,attribute,version> triple) but often it has not.

2. EC.IO

(1) We considered five alternatives for handling retries of unsuccesful I/O operations:

(a) having two different commands for these two cases: one that retries, either once or until it succeeds, and one that instead of retrying returns a failure indicator;

(b) having a parameter on the command specifying how often to retry, and having the command return a failure indicator;

(c) having a failure indicator, and having the user program try again if it needs to retry transmission;

(d) having a special "retry" command, with a label operand, which the user can call to have the I/O command with the specified label retried.

(e) omitting the failure indicator for the data items where it is not currently used.

The first and fourth alternatives yield a more complicated interface than the third and provide no extra capability. The second results in extra (non-machine-dependent) programs in the EC. The fifth alternative would build knowledge of the application into the EC. The third alternative relegates decisions about retrying to the user programs, and we chose this one.

(2) We have considered four alternatives for handling the discrete inputs and outputs.

(a) Treating input and output differently, allowing user programs to use a READ command to read in entire discrete input words, but providing a special WRITEBOOL command so that user programs could write individual bits appearing in the discrete output words.

(b) Adding a READBOOL command that would read in a discrete input word, pick out the bit for a particular discrete input data item, and return it as a boolean value. Alternative 2 was rejected because not all the data items in discrete input words have boolean values. For example, /IMSMODE/ has five values, one for each switch position.

(c) Provide the user programs with a way to specify a range of bits within both a discrete output word that they want to write out and within an input word, so that they can request individual discrete inputs in a symmetrical fashion. Alternative 3 leaves some of the responsibility for non-interference between discrete outputs to the device interface modules, since they must specify the correct ranges.

Current: All of the above alternatives were based on the decision that the EC would sometimes identify outputs and inputs by class name rather than the individual data item name. This was done both for efficiency reasons and because it was believed that knowledge of the location of a data item within a discrete input or output word was device dependent rather than computer dependent. A much more consistent interface is achieved by always using the data item name. The EC implementer is now responsible for knowing the identity of a TC-2 I/O item, but not responsible for knowing its meaning. The efficiency problems are resolved by allowing a single command to take a list of parameters so that the EC implementation may perform operations to a single I/O word simultaneously rather than sequentially. This also eliminates special treatment of double data items.

(3) We originally designed the reading of intermittent data with an access function that indicated whether or not the data were available and an undesired event if a user program tried an intermittent read operation when the data were not available. This seemed dangerous, since a slight timing difference could cause an undesired event, and the user programs could not avoid the UE. Instead, we have chosen to allow the read command at any time. If the data are not available, the success indicator returns *false*. This is consistent with our general policy that it should be possible to avoid UEs by correct programming.

Because the intermittent data is read just like any other, we decided not to have a separate command name for it.

(4) We considered having serial inputs identified by class names rather than by individual data item name. Interpretation of the identification bits was considered the responsibility of the associated device interface module. We decided that identification of the data item is an EC responsibility, but interpretation of the item remains the responsibility of the DIM.

(5) Note that sometimes an output should go to more than one data item. We originally handled this by letting users repeat sets of parameters to i/o commands. and saying that the order was unspecified. Since we no longer have i/o commands per se, but rather use assignment (and other bitstring) operations, we have expanded our general assignment statement so that many sources and many destinations can be given at once; the assignment happens in an unspecified order.

(6) We promise that an output transmission will occur when an enabled output data item is used as a destination. We do not say when an input transmission will occur. This is because we can get away with it in the latter case, but not in the former (because an output transmission has visible effects). We hide when input takes place because someday there may be direct-memory-access input, and the computer really won't be able to control when an input item changes value.

(7) We did not include the names of the data items in the main document, because we wanted to emphasize the fact that the architecture of the Extended Computer's i/o operations doesn't depend on those particular names. If the design of the Extended Computer were used with the TC-2 for some other application, the names of the data items would not be part of the technology transferred.

(8) We chose special names for the data items' bitstring spectypes because we felt that the representation for each was likely to change in the event of a device replacement, and probably wouldn't be the same as that of a non-data-item bitstring anyway. For instance, we might choose to represent "normal" bitstrings as contiguous and left-justified within a word, but we clearly don't have this option with most of the data items (see //FPANGL//, for instance).

(9) The signal converter is tested by sending particular values to it and then reading back the results of the internal signal converter manipulation on the values. The proper relationship between the values sent out and the value read in can be characterized by a set of equations. The design issue is how much of the knowledge should be hidden within this module: both the equations and the choice of test values, just the equations, or neither. The equations are based on the behavior of the channel, and therefore belong

within this module. The choice of values could be considered part of the software requirements; they affect the displays seen by the pilot, and are documented in section 4 of the requirements. However, the choice of these particular values is partly influenced by hardware characteristics. Further, if they are not hidden, the interface to this module becomes much more complex. We have chosen to hide all of the information even though it means hiding some details about the required functions in this interface. We assume that the test values are likely to change with the hardware and not for any other reason.

(10) We decided to hold user programs responsible for avoiding interference between the diagnostics and the regular commands rather than build monitors into the I/O commands and diagnostics. The diagnostics are not expected to be run when the software is doing anything else. Monitors impose a run-time cost in the regular commands.

(11) We formerly had an access program parameterized by data item that returned a boolean telling whether or not the last transmission of that item was successful. We replaced the program with built-in boolean variables, because doing so lets the user avoid an unnecessary store operation in some cases. One such case is when the user wants to use one of the builtins as a source and a <NOSTORE...> operand as the other source. There may be others.

(12) We can generate more efficient code if we know beforehand which *x*_SUCCESS items will and will not be accessed. We could have the user declare which ones he plans to use, but our translator software can easily discover that anyway. Further, if the user gave an incorrect list, that would be a UE that our translator software would have to detect anyway.

(13) We could make it a UE for a user to assign a value to one of the *x*_SUCCESS booleans. However, the user could do that via a pointer, and we wouldn't detect that. Assigning a value to the variable does have consistent (although not very useful) semantics, and we decided not to prevent it.

(14) EC refers to the Requirements Document for the definition of its i/o items. Some items are specified in that document to be always preceded by a 0 bit. If we changed the defintion of those items (either in the Requirements document or in this spec) to make that 0 bit part of the item, we could achieve some efficiency gains in our implementation. For one thing, we wouldn't have to issue instructions to zero out that bit. This didn't occur to us until fairly late in the project; for now, we choose not to go back and take advantage of this ‘‘trick"; however, we may decide later that we need the efficiency bad enough to go back and re-define our i/o items. This design issue is to record the possibility, and the fact that we wish we had thought of it sooner.

3. EC.MEM

(1) We considered dividing memory into banks that would be tested separately, allowing partial rather than complete shutdown. We decided not to do so at this time because the system lacks the ability to exploit it and we could do so easily in the future.

(2) A previous design implied that invoking the access program associated with a test actually started the test. Because future computers may have tests ongoing, or running in the background, we changed our design to indicate that invoking a program merely returns the most recent result of that test. If a future computer is *required* to start a test at a certain time, we can add start-test commands later. Returning the value may take a substantial amount of time in some cases. The major change this caused was in the case of the memory test. Before, there was a command to start the test, an event signalled when it was done, and a program to retrieve the result. The motive was that the invoking program would want to do something else while waiting for the test to be completed. However, some program would have to wait idly for the event to occur anyway, and so we lose nothing by letting the memory test program just take a long time to return. We gain a uniform interface, with no special cases.

4. EC.PAR.1

(1) In earlier designs of this interface, timing constraints were associated with specially designated blocks, implying that these blocks were the scheduling units. The process mechanism was unnecessarily complicated, put too many restrictions on the internal structure of processes, and gave away more information than the one here.

(2) We considered having START and STOP commands so that one process can explicitly affect the ready/waiting state of another process. The problem with a STOP command is that a process cannot be safely stopped at any arbitrary point. We fixed this by adding "homing points", but specifying homing points also cluttered up the algorithm descriptions. So we dropped the idea, relying on more conventional synchronization mechanisms instead.

(3) Earlier versions made an outer "do forever" loop implicit. Thus the process would execute a "INIT" block once whenever the process was started and then repeatedly execute a "FREQ" block until the process was stopped. We have decided not to include an implicit loop because we do not want to limit the internal structure of the processes. Also, the process would be easier to read if all the control was shown explicitly. Process bodies can now be specified just as subprogram bodies are, making the overall specifications of the Extended Computer simpler.

(4) At one point we had intermittent processes wait for a start event and then run until a stop condition existed. We found it simpler to define a single boolean and have the process pass its start point only when the boolean was true. This eliminated the need for the event interface in the EC and eliminated ambiguous cases such as the start event ocurring when the stop condition held.

(5) At one point we had a special class of processes called init processes. We recognized these as a special case of Demand processes and decided to simplify the interface by exploiting that fact. This allows some processes to be used both as init processes and under other conditions.

(6) It is possible for a programmer to write a process that runs out of statements to execute. We considered three alternatives for demand processes:

(a) Stating that it is an undesired event for a process to finish, i.e., making it a requirement that each process contain an infinite loop;

(b) Assuming that a completed process is in the ready state, but that it has a null statement list to execute if it becomes running;

(c) Assuming that a completed process is in a waiting state, waiting for an event that will never occur.

We rejected (a) because it builds too much information into the Extended Computer and it is an unnecessary restriction. We rejected (b) because there is no point in having a completed process compete for a processor. Alternative (c) is a reasonable compromise for the Extended Computer interface. If it is considered undesirable to have completed processes, this should be prohibited by programming conventions.

For periodic processes, it is a UE because we introduced the way for the writer of the process to turn it on and off.

(7) We have decided not to include relative priorities for the different processes because fixed priorities do not generally work when there are real-time constraints.

(8) In an earlier version, we had no distinction between periodic and demand processes because a periodic process can be viewed as one that waits for a particular stimulus, i.e., the passage of a particular amount of time. However, one of the timing parameters needed for periodic processes is not useful for demand processes. In addition, periodic processes must have restrictions on the synchronization operators within the periodic loop because the indeterminate wait associated with synchronization operators makes it difficult to prove that the loop can be scheduled regularly as required.

(9) In an earlier design, we did not explicitly distinguish intermittent periodic processes. We now distinguish them in order to increase the likelihood that we can take advantage of the intermittency in the scheduling of processes. Earlier we distinguished them by calling them intermittent, now we use the presence of the ON/OFF parameter to distinguish them.

(10) We considered specifying periodic processes in terms of frequency rather than in terms of time intervals. Because we wanted to specify the deadline as an interval, we decided it would be more straightforward to use two intervals. These two parameters adequately constrain the variations in regularity.

(11) We have an undesired event assumptions that says there won’t be too many demand processes for a periodic process to miss its deadline. The assumption is worded with that orientation because it is impossible to tell how often a demand process must run.

(12) We used to allow the body of a process to be any statement list. We now restrict it to a call on a previously-declared program. In this way we maintain a clear distinction between process and program,

and therefore allow future extensions to include run-time creation of processes without run-time creation of programs, vice versa. The restriction does not restrict what we can do with the current version; it merely paves the way for future extensions.

(13) Previously, there was a parameter in the process definitions with which the user specified the maximum CPU time required by his process. We removed this from the high-level EC interface because (a) the user doesn't have enough information to provide it; (b) the information is machine-dependent; and (c) the length of time an operation takes can vary greatly, depending on the storage and representation of the operands, for example. This is information is now provided to the EC implementation at a lower level, where the processes are divided into scheduling blocks; it may be provided by software that examines the code, or it may be done manually.

(14) Why do periodic processes need both a semaphore to begin and an on/off boolean to start/stop them? The semaphore is there in order to allow us to stage the initialization. During initialization we may want some of the processes to complete their initial sections before others begin. We do that by letting them signal the starting event for the others. We may want a process to initialize but not start its intermittent execution. If we tried to use the boolean to do that there would be critical timing about the question of when to turn it off. Once the starting event becomes non-negative, it is never looked at again. (DLP 9/24/85)

5. EC.PAR.2

(1) Regions with an exclusion relation were selected for Extended Computer synchronization primitives because

(a) they allow concurrency constraints to be expressed directly rather than as an implication of run time synchronization;

(b) they express the exclusion relationships in a form that can be interpreted efficiently by a pre-run-time scheduler;

(c) there is an algorithm for generating run-time synchronization from the exclusion relations;

This is the simplest acceptable alternative. Rejected alternatives included:

(a) disabling interruption: once an identified section of code starts executing, it must run to completion. This alternative was rejected because it is prejudiced toward a single processor: it overly restricts the parallelism by stating that no other actions can be taken simultaneously with the code section, rather than specifying which other actions may not be taken;

(b) simple mutual exclusion: specifies all exclusion relations as equivalent, i.e., a section of code that excludes any other excludes all others. This alternative still places too many restrictions on the parallelism because many of the identified code sections need not exclude each other.

(c) named regions with mutual exclusion. Rejected because it assumes that the exclusion relation is symmetric.

(d) exclusion via synchronization primitives: using synchronization primitives such as those in EC.SMPH to effect mutual exclusion. Rejected because (1) synchronization primitives that are being used for other interprocess synchronization or communication purposes cannot be distinguished from those used for exclusion without additional commentary, (2) the exclusion requirements are implicit in a solution based on synchronization primitives, rather than stated explicitly as they can be with identifiable regions, and (3) the exclusion information (implying scheduling constraints) is embedded in and scattered throughout the text. These properties of the synchronism primitives make it difficult to do pre-run-time scheduling without substantial preprocessing.

(2) Many useful forms of synchronization were rejected for the Extended Computer because they do not depend on the implementation of parallel process. Application-oriented synchronization operations may be developed using the exclusion relations, and semaphores.

(3) Can a region be excluded from itself? Is that useful? Yes, because in the case of non-reentrant code, this is how we will probably prevent disastrous re-invocations.

(4) We adopted the REGION/END_REGION syntax (as opposed to a single REGION program that takes a statement-list as a parameter) because we did not want to require that regions nest.

6. EC.PGM.1

NOTE: Design issues 1 through 7 refer to a previous control structure we had included in the EC which is documented informally in [APC]. That control structure is no longer a part of EC, because we concluded that it required too much sophistication in implementation, and that simpler constructs hid the hardware characteristics just as well.

(1) Alternatives considered for the syntax of a guard are shown below.

(a) Boolean variables or constants only. All the boolean variables must be assigned values before the limited program is executed.

(b) Any sequence of statements assigning a boolean value to a special guarded command register.

(c) Allowing a limited program list as a guard.

(d) Allowing a program to define the value of a guard (defined guards).

(e) All of the above.

We chose (e). The semantics can easily be defined formally [ITTI2]. Defined guards save code by avoiding duplication of statement lists, which would otherwise be required because of syntactic limitations.

(2) We chose to have the Extended Computer provide the IT-TI construct rather than the more common IF-THEN-ELSE, CASE, and DO-WHILE constructs because IT-TI serves for all purposes. It allows some programs to be written as one loop that would otherwise require several, thereby saving variables and predicate evaluation. IT-TI has a mathematical semantics that allows systematic construction of the program's function [ITTI1].

(3) Dijkstra's guarded commands are nondeterministic; of the true guards, only one is selected, but there are no rules defining which one is selected. We chose a deterministic construct because they allow simpler guards.

(4) We considered providing a FOR command (FOR I = 1 to 10 DO...) but decided against it because a. the same purpose can be served with the IT-TI command; b. many special cases and questions arise with the FOR command.

(5) We considered having an implicit final LP of the form (true,SKIP). We decided not to do this in order to encourage the programmer to consider every case carefully.

(6) Should statement lists be allowed to contain declarations, and what is their scope? We decided that it was harmless (from this module's point of view) to allow it. The scope of all declarations is global and items must be declared before they are used, but these are issues belonging to the EC submodules that provide the declarations.

(7) In an earlier version we allowed Dijkstra's *cor* and *cand*. We have eliminated them because the same effect can be obtained with the use of defined guards.

(8) It is not a UE for an !!exit connector!! to contain the same exit twice (as long as no conflict results), nor for an !!exit connector!!'s !!interval!! to specify an empty range. Both conditions are harmless, and may result normally from automatic code generation.

(9) The ++PAR++ operation has been added, deleted, and added again to the EC. It was originally part of the it-ti control construct, no longer used. When it-ti was removed, so was ++PAR++ because we believed that we could accomplish the same objectives with destination lists and operand lists. However, ++PAR++ allows us to make one kind of optimization that we couldn't make before — namely, allowing more than one data item that lives in a single TC-2 word to be read/written with the same instruction. There may be other re-ordering optimizations of which we are not yet aware. To make the user exploit this capability, we need to tell him (in the Efficiency Guide) that he should batch his reads and writes, if the items live in the same word. Otherwise, doing so may actually be more expensive because it will force him to use temporaries (rather than NOSTORE). That means the user of this implementation of the

EC has to know what items live in the same word. However, that information will *not* allow him to write incorrect programs even if that information should change, and so our abstraction is preserved.

(10) It is a UE to introduce a name for a label that is already in use as the name of some other kind of object, but not an error to re-use label names in different constructed programs. The reason is that it is harmless, and may be convenient for our users. When a label appears in an exit-connector, it of course refers to a label within the same constructed program since we do not allow "inter-program" branching except out of an exit and into an entrance. Therefore, labels are naturally "scoped" and re-using label names in different programs is harmless. We could easily have decided this issue the other way; it doesn't appear to matter much.

7. EC.PGM.2

NOTE: Design issues 1 through 3 refer to a design that allowed user-defined programs to have parameters. Because the semantics of parameter passing do not depend upon the hardware, we concluded that users could employ other facilities to achieve a parameter protocol (e.g., assignment before and after a program call).

(1) Should actual and formal parameters be specified by type class, specific type name, or using type attributes such as range and resolution? We decided that type agreement should depend on the specification chosen by the programmer. Other alternatives would force us to write separate programs for each specific type or to include a parameter passing mechanism that would be more general than needed for most cases.

(2) We added PARM_GIVEN because programs must be able to tell if an optional parameter was supplied or not. We thought about making it a built-in value that any type of variable could take on; then programmers could ask, e.g., if p1=PARM_GIVEN. However, because output parameters can be optional, we didn't want programmers checking their "value".

(3) Programmers need not supply trailing commas when optional parameters at the end of a parameter list are omitted. For example, instead of +pgm1+(a,b,,,) one may write +pgm1+(a,b). Besides the obvious convenience, this will allow us to add optional parameters to the end of any access program parameter list, yet not force all calls on that program to change.

(4) We added the feature of ranking programs' access speed because the current computer has the capability of doing fast subroutine linkages in certain areas of memory. Because a replacement machine may not have such a capability, we made the relationship "not-slower-than", which we can trivially implement by doing nothing. We make no firm promise about the ordering, however, because we recognize that we cannot make access to a subroutine not-slower-than access to an expanded macro that simply lives in-line.

(5) We considered giving the user the ability to specify whether a program was to be invoked by subroutine linkage or in-line (macro) expansion. Howver, we realized that macro-expansion can be done independently of the host machine, and hence is not an appropriate EC facility.

(6) We included the facility for program loadcons because it is theoretically possible to key in a new value (body) for a program after load-time. However, such a facility would only be useful if (a) the new program took exactly as much space as the one it replaced; or (b) there was plenty of extra space available in the computer. Since (a) is machine-dependent and (b) is not true, we have not implemented program loadcons in the Extended Computer.

(7) The Extended Computer provides both program variables and pointers to programs. Our current application does not require both, and so only program pointers are implemented. Both are provided in the design because of the usual analogy between programs and data. However, the difference between a data variable and a program variable is that we provide operations to create new values of data variables at run-time, whereas there are no such operations (e.g., composition, union) for programs. If our data had a small number of values, all known at sysgen time, we could program only with data pointers and no variables. While that is not the case for data, it is the case for programs.

(8) We briefly flirted with the idea of not allowing users to use UEs as a !!source!!, or in a <REF...> or <DEREF...> construct. For a while, doing so was an unimplemented feature. The reason was that if a UE was used in such a way, the user's program would behave differently on the production EC (in which

the UEs are all removed) than it would on the check-out EC. However, that assumed that we would always remove ALL UEs (i.e., all built-in pointers to UE-handling programs). We realized that we could easily keep the ones that the user needed. The user knows that in the production EC, no UEs will be raised, and so we assume he is smart enough not to use one of the built-in UE names in such a way as to force us to keep it unnecessarily. This decision allows us to preserve two important principles -- a program that works on the checkout EC will work on the production EC, and a built-in object looks just like a user-defined object of the same kind.

(9) We thought about requiring all programs to be given using the !!typed literal!! format, but realized it wasn't necessary. As the initial value of a declaration, the spectype is given. When invoked directly as a literal, the spectype information isn't used. When used as a !!source!!, the spectype can be determined from the spectype of the destination.

(10) It is not a UE to use the same entrance or exit names in different program spectypes. We thought it might be convenient in some cases. It is also not an error is a program fails to include code to branch to all of the exits associated with its spectype. Therefore, the user is able to define two spectypes, the exits of the first of which are a proper subset of the exits of the second. This will allow the user to build some upward compatibility into his programs because programs of the first spectype may be substituted for programs of the second. This might be useful. However, we could easily have decided to make all entrance and exits names unique; it doesn't appear to matter much.

8. EC.PGM.3

(1) The UE "constant destination" comes in both sysgen and run-time flavors. It is a run-time UE because a destination can be given via a pointer, and we cannot tell at sysgen time whether the pointer points to a constant or not. It is a sysgen-time UE because sometimes we must guarantee to detect it at sysgen time (e.g., for ++SET++).

(2) There are several ways to interpret the effects of system generation programs. Since we don't allow exit-connectors after a sysgen program invocation, it is clear that they are executed before run-time in the order encountered in the source code. However:

(a) they might be executed before any code is compiled; thus, the values used in the compilation would be the last ones computed in this pre-run-time execution. This has the effect of making all but the last ++SET++ of a particular variable meaningless, for example.

(b) they might be executed while the code is being compiled. Thus, the values used in compiling the code would be the "current" values; i.e., the values as most recently set by the sysgen programs.

(c) they could be executed after the code is compiled, having no effect on the compiled code at all.

Clearly (c) is the least useful. In our opinion, (b) is the most useful. However, for a long time there was an ambiguity in our specification that went unnoticed until the Extended Computer was implemented. The implementation team interpreted the spec in the manner of (a). If we had more time, we would change our implementation (and our design) so that (b) was the case. Note that such a change could produce an "upward compatible" EC; that is, using programs would still be correct if they were careful only to ++SET++ values once.

9. EC.SMPH

(1) We originally had more complex synchronization operators that met many immediate demands of our application. As we prefer the Extended Computer to be as application-independent as possible, we chose synchronization operations for the Extended Computer primitives that would be as simple as possible, but that could be used as building blocks for more specialized synchronization operators. For the more complex synchronization operations, see the specifications of the Application Data Type module [ADT].

All of the following alternatives for the Extended Computer synchronization operations were rejected either because they are more complex than the operations selected or because they can be built using the operations selected.

(a) P and V operations on semaphores;

(b) eventcounts [REED79]: Also rejected because we weren't sure we would need them;

(c) P and V supplemented by eventcounts;

(d) UP, DOWN, and PASS supplemented by event variables. A simple generalization of event variables, event-booleans can be implemented in the Application Data Type module in a machine independent way.

(e) V, DOWN, PASS, and eventcounts.

(2) At one point, we provided a semaphore-to-integer conversion program. These were deleted when we could think of no reason to use it. If such a need arises, it would be a straightforward extension, allowing upward-compatability between programs written now and later.

(3) This used to be a submodule of EC.PAR, because semaphores are used to synchronize processes. However, the secret of implementing them has nothing to do with processes, so semaphores belong in a module of their own.

(4) An earlier version implied that an +UP+ operation would awaken the processes in turn and each awakened process could execute, possibly downing the sempahore, thus blocking processes not yet awakened. This is inconsistent with the formal definitions and would make programming more difficult (Which process would +DOWN+ the semaphore? Also, this would have forced the use of regions to make sure that all waiting processes were allowed to proceed after an +UP+.) The footnote that created this impression has been removed. The binding definition is still the formal one, which holds that all processes in the middle of a +PASS+ operation for a semaphore get to proceed.

(5) Should the EC decrement a semaphore that it just incremented to signal an event? We decided so for a brief period, because it is not clear who else will if the EC does not. However, when a raised semaphore gets decremented turns out to have subtle effects of when programs run. Which scheme you get is really machine-independent. So we changed back to the old scheme, in which the EC merely increments the semaphore; a user program must decrement it if it ever wishes to await on that event again. However, this is one of those issues that refuses to die, and the next design issue discusses our current approach.

(6) We would like for programmers to be able to write code for processes that wait for an event (the semaphore to be incremented) without needing to assume anything about other processes in the system. Similarly, we would like to be able to write the signalling program inside the EC without needing to know anything about the processes that might wait on it such as how many such processes there may be. Thus we exclude solutions such as providing a different semaphore on the interface for each waiting process or doing one up-operation for each process that might wait, etc.

It turns out that there is no simple solution written entirely in terms of the d-operations (up/down/pass) with these properties. The solution we used in the EC was to up a semaphore to signal an event. Processes using the EC must had to down the semaphore to have it signaled again. This simply moves the problem outside of the interface of the signaling module so the works are accessable. Processes must still cooperate to use the event (decide who downs the semaphore) and additional synchroniztion operators must be created.

The problem is that we want to release all of the processes waiting on the semaphore (executing a pass) without also allowing processes to proceed that were not waiting at the time the event occurred (including processes that were originally waiting and have looped around and are executing the wait operation again).

Although there is no solution strictly in terms of d-operations, the problem can be solved by providing a region R on the signaling module's interface associated with each semaphore S used to signal an event. The module guarantees that the event will be signaled by a single up operation on S follwed by a single down operation and that both operations will occur inside the region R. The region R can then be used to prevent processes from executing a pass at the wrong time.

A process that wishes to wait for the event inside a loop and execute the loop exactly once for each occurrence of the event must define a region R' and an exclusion relation such that R excludes R'. As long as the pass(S) is executed only inside the region R, it is guaranteed that only one pass will be executed for each event.

Note that a process can also execute the pass outside the regions if it wants to continue as long as the semaphore is positive. This form is also useful, and corresponds to our "=T" events on other interfaces. (Stuart Faulk, 4/23/87)

We disallow user regions excluding EC built-in regions so that no user can effectively disable the EC event-signalling mechanisms.

10. EC.STATE

(1) The following transitions are not included in this interface for the following reasons:

off to failed:	not relevant to user programs;
failed to off:	user programs cannot respond to anything
operating to off:	when the computer is off;

Note that failed to operating does not occur with the current computer; it must be cycled through "off" to get back to operating from failed. However, future computers may make this transition possible (perhaps by re-booting), and so this transition is subsumed by the definition of power up.

(2) There may not always be a grace period after @T(!+failed state+!). Two alternatives were considered: to leave out the grace period altogether, or to include it as a system-generation parameter. We selected the latter to allow for future use of an improved computer.

(3) How do we distinguish between malfunctions that user programs must detect and handle (possibly by calling +S_FAIL_STATE+) and malfunctions that are detected inside the Extended Computer? Malfunctions are detected by this module if they are reported by the computer without software action; for example, malfunctions signalled by interrupts. Whenever a malfunction is detected because of an action dictated by the requirements, such as a diagnostic test, detection is left to a user program. The malfunctions described in various test programs (EC.PAR.1, EC.IO, EC.MEM, EC.TIMER) belong to the latter category; all others, the former.

(4) Future technology may make our three-state model appear oversimplified, because a system may have degraded states: that is, states without the full capability of "operating", yet not dead in the water like "failed". A degraded state may occur in a single-processor system, or in a multi-processor system where one or more processors have ceased to operate. It is important that acquiring this capability results in adding to (not revising) the present specification. We cannot add a degraded state now because we cannot implement it, and programs depending on it would be not be correct. However, we can plan for the addition by assuming that there are "*at least* three states", etc.

(5) Which module is responsible for the close-down procedures? We decided that any shutdown action that is required for every computer failure and is computer-dependent should be done by this module. If the action is device-dependent, such as setting the bomb-release output to a safe value, it should be done by the device interface module.

11. EC.TIMER

(1) In earlier versions, we had clocks and timers; clocks counted up and timers counted down. They were completely distinct from timeint entities; they were declared separately and had their own set of operations. We removed the distinction as the interface grew and grew, and we realized that it would be both useful and consistent to let a clock/timer do most anything that a timeint can do.

(2) We considered providing only clocks or only timers (in the sense of issue #1). Clocks are useful for measuring elapsed time; timers are useful for detecting the end of a previously specified time interval. We wanted the capabilities of both because otherwise user programs would have to use one to simulate the other. This would lead to inefficiency and possible duplicate efforts especially on a computer that provided both.

(3) We considered having this module offer a special "waittime" command, instead of using the general semaphore mechanism. There seems to be no advantage in using a special mechanism for timed events.

(4) We considered treating the following actions as errors:

- starting a running timer,
- setting a running timer,
- stopping a non-running timer,
- reading a non-running timer,
- stopping a timer that has run down,
- reading a timer that has run down,

but these actions are not necessarily senseless.

5. We considered having a timer signal a UE if it runs past its capacity. To have it start over seems the most useful. Further, a timer might run past its usual limit, for no fault of the software. In contrast, setting timer with too large a value is a clear software error. Therefore we made exceeding the maximum capacity an undesired event in a set operation.

(5) In an earlier design, there was a single maximum capacity for all clocks and a single maximum for all timers. It was pointed out that clocks and timers are used for very different purposes, some for measuring very small changes over a small period of time, and some for keeping track of a long period of time, with less concern for small changes. In order to achieve this flexibility without undue use of resources, we decided to allow programmers to specify capacity and minimum measurements for individual timers and clocks.

(6) In an earlier version, clocks could only be set to zero, but this seems unnecessarily restrictive. One of our reviewers (D. Hill) said "I believe we may need a +SET_CLOCK+ for clock corrections or for time-of-day clocks." The restriction went away when we merged timers and timeints.

(7) Timers used to be a submodule of what was called the "Sequential Execution" module, because programmers would presumably want to transfer control based on the value or action of a timer. However, the secret of the timer module has nothing to do with flow of control, and so it became a module of its own.

APPENDIX B

Implementation Notes

1. EC.DATA

(1) If the user provides a subrange assertion, the information may be used to reduce the amount of operand shifting necessary before an operation takes place.

2. EC.IO

(1) The part of the I/O submodule that handles the relation between data item names and TC-2 instruction sequences should be a sysgen time program and should be table driven. It should be organized into submodules in accordance with the structure of the Device Interface Module, because changes are likely to be concentrated on individual devices.

3. EC.MEM None.

4. EC.PAR None.

5. EC.PGM.1 None.

6. EC.PGM.2

(1) This module does not determine where programs are located in memory. It uses programs in the memory allocator module to request space.

(2) This module uses the System Generation module to do assembly-time parameter type checking.

7. EC.PGM.3 None.

8. EC.SMPH None.

9. EC.STATE None.

10. EC.TIMER None.

APPENDIX C

Assumptions Lists

BASIC ASSUMPTIONS:

1.1. EC.DATA

(1) The Extended Computer can provide a way to compute a result without storing it into a declared variable. Access to this result will usually be quicker than access to other variables.

(2) The attributes of a value will be known whenever a variable is used as a source or a destination. If the attributes specified for the variable when it is used as a source are not the same as were specified when its value was determined, the result may be any value.

(3) The Extended Computer can store numeric quantities with any desired range and resolution. It can be expected that (a) variables with a small range-to-resolution ratio will require less actual memory space than variables with a large range-to-resolution ratio, and (b) that operations on such variables will be faster than operations on variables with a larger range-to-resolution ratio.

(4) Range and resolution are adequate characterizations of a numeric variable; i.e., the needs of an application programmer can be adequately expressed by a lower bound, upper bound and guaranteed resolution.

(5) The Extended Computer can store bitstring quantities of any desired length, up to a limit known at system generation time. Longer bitstring entities may require more storage than shorter ones. Operations on longer bitstring entities may require more computer time than operations on shorter ones.

(6) Whenever a numeric value is stored into a variable with a resolution different from the source, the value stored should always be the closest value that can be represented in the destination. The programmer need not specify the conversions to be made; the best choice can be made by the EC implementation.

(7) There is no need for operations that allow a bitstring value of one length to be assigned to a bitstring variable with a different length.

(8) The operations needed for calculating new numeric values are addition, multiplication, division, subtraction, absolute value, complement and conversions.

(9) Division may result in a loss of all significance. This danger cannot be hidden entirely from the programmers, since they may have information that can be used to choose safe, efficient algorithms. The following division options are sufficient:

a. The quickest division can be performed if the programmer provides an upper bound for the result. The better the bound, the more significance is preserved. If the bound is too low, all significance may be lost.

b. A slower algorithm can be used if the programmer cannot provide an upper bound.

c. If the programmer cannot provide an estimate of the maximum result and prefers to avoid the expense of the slower algorithm, the Extended Computer can determine whether or not division can be safely performed. The EC can return the sign of the quotient even when the operation cannot be safely performed.

(10) For any variable, it is always possible to implement a uniform resolution over the entire range of that variable.

(11) Whenever the program compares two numeric operands for equality, programmers need to define a threshold, such that if the difference between two numbers is less than or equal to the threshold, the

numbers are considered equal.

(12) It is acceptable for the results of an operation to have a larger resolution than the resolution of the destination. The approximations needed to store the result can be assumed to be acceptable for the application.

(13) Only four kinds of entities are needed: variables, which can be changed at any time; ascons and literals, which can be changed by reassembling the program; and loadcons, which can be changed when the program is first loaded into the computer but not while it is running.

(14) The following operations are sufficient for efficiently producing new bitstring values from existing bitstring values:

a. AND, OR, NAND, NOT, MINUS, and XOR, defined in the usual way, operating on corresponding bits in two operands of equal length;

b. SHIFT operation: A bitstring is shifted either right or left a specified number of bits with zeros shifted into positions vacated by the shift;

c. REPLACE operation: A portion of a bitstring is replaced by the value found in an equal-length portion of another bitsring;

d. CAT operation: A bitstring is formed by concatenating two previously existing bitstrings.

(15) If the result of converting a real to a bitstring has more bits than the bitstring operand, the bits to the right of the rightmost bit of the destination bitstring may be ignored.

(16) Arrays with dimensions that vary at run time are not needed in avionics applications.

(17) Avionics applications do need arrays in which the type class is real, and the elements are variables with attributes that may vary independently of the attributes of other elements of the same array.

(18) Arrays in which the indices are not a contiguous subset of the integers are not needed in avionics applications.

(19) Avionics applications need to take advantage of any capability that the computer has to allow faster memory access to certain data. The Extended Computer can implement a "not-slower-than" relation for any two declared entities x and y, so that x will be accessed no slower than y. User programs can determine desired rankings at system generation time; it is not necessary to change the rankings at run-time.

1.2. EC.IO

(1) The only information needed by user programs to identify inputs or outputs is the data item name given in the requirements document [REQ]. It is possible to characterize all transmissions between the Extended Computer and its associated hardware as either input or output.

(2) Input data items and output data items are bitstring entities. Some can only be used as a source in a statement (read-only); some can only be used as a destination in a statement (write-only); some can be used as either (read-write). No input data item is write-only. No output data item is read-only.

(3) It is possible to turn off (disable) input/output transmissions. A disabled data item has no effect on and is not affected by the external environment.

(4) No application program will need the identity code and subitem identifiers in Serial Input Register Data (see [REQ]).

(5) It is possible for the software to determine the success of I/O operations. (Of course, this assumption is obviously false if we consider hardware failures. However the correctness of our software is contingent on that assumption.) An unsuccessful operation may not change the value of the associated data item.

(6) Some input data items are only available intermittently and the EC can notify user programs when new values for such data become available.

(7) Each i/o operation can be guaranteed to complete within a fixed period of time. This worst-case timing requirement varies among data items; the time associated with each data item can be determined at

system-generation time.

(8) Each channel diagnostic program may interfere with a specified subset of the input/output commands. They will not interfere with any other commands.

(9) Use of either the discrete diagnostics or the accelerometer-torque diagnostics may cause the IMS to lose its alignment and velocities (i.e., have the same effect as disabling the IMS temporarily).

(10) The following aspects of the input/output can be tested independently:

the AC aspects of the signal converter channel,
the DC aspects of the signal converter channel,
the cycle steal channel A and serial input channel 1,
the cycle steal channel B and serial input channel 2,
discrete input word 1 and discrete output word 1,
discrete input word 2 and discrete output word 2,
discrete input word 3 and discrete output word 3,
the IMS gyro torque registers and the accelerometer accumulators.

1.3. EC.MEM

(1) A memory diagnostic program can check whether portions of memory are reliable. This program does not interfere with other programs. The test may take a substantial amount of time to complete.

1.4. EC.PAR.1

(1) Processes (executions of programs) may execute in parallel with no restrictions on their relative speeds, except where they are explicitly synchronized with each other (see EC.PAR.2).

(2) The number of processes need not vary at run-time. It may be set at system generation time.

(3) All demand processes can start when the system is turned on (i.e., when @T(!+power up+!) occurs); some will perform initialization routines; the remaining demand processes will wait for a semaphore to become nonnegative.

(4) The process mechanism will be able to detect the event @T(!+power up+!).

(5) Processes are not called as subroutines by other programs and do not return control to other programs.

(6) We need only distinguish two process states: *active* or *suspended*. Only an active process can progress. A suspended process is ineligible to progress (continue execution).

(7) The state of a process changes between active and suspended only when it uses the process synchronization mechanisms described in sections EC.PAR.2 and EC.SMPH or when it has executed the last statement in its body.

(8) All processes are either periodic or demand and exist throughout the life of the system;

The bodies of periodic processes are to be executed at regular intervals (their period). The period of a process may change during system execution. A periodic process may be suspended when a specified boolean variable is false and start again when it is true.

Demand processes wait for a semaphore to be nonnegative. They should be executed each time the semaphore is incremented. They will decrement the semaphore once per execution.

(9) Demand processes can be adequately characterized by specifying the values of two timing parameters: maximum CPU time requirement and deadline for completion.

(10) Periodic processes are adequately characterized by three timing parameters: maximum CPU time requirement, deadline, and period.

1.5. EC.PAR.2

(1) User programs may contain contiguous sections or *regions* of run-time-executable statements that may not be executed concurrently. These concurrency constraints can be expressed in terms of an exclusion relation on the regions, i.e., where region 1 *excludes* region 2 if region 2 may not start while region 1 is executing.

(2) Regions may overlap other regions or be embedded in other regions.

1.6. EC.PGM.1

(1) The only sequence control constructs needed are those that choose a path based upon the results of the invocation of a program.

(2) The number of entrances and exits of a program is finite, and the upper bound can be determined at system generation time.

1.7. EC.PGM.2

(1) Some program entities should be invoked faster than others. Such a relation will not depend on when the programs are invoked; the relative ranking can be determined at system generation time.

(2) It is not necessary to provide users with the capability to create programs that take parameters. Other mechanisms available to him (such as assignment before and after the "body" of the program) suffice.

(3) It is necessary to provide facilities for recovery if a programming error is detected by a program during execution. It is up to the author of a called program to determine what programming errors his program can detect; it is up to the caller of a program to determine the action that should be taken if one of those errors occurs. It is not necessary to pass parameters to the recovery program.

1.8. EC.PGM.3

(1) If a program will be reentered while already in use by another process, it is the responsibility of the programmer to make sure that local storage is saved and restored as needed. EC programs are not automatically provided with new storage when they are reentered.

(2) There is no need for a mechanism to allow programs to cause the calling program to resume execution anywhere else than immediately after the call.

(3) The identity of a data entity that is passed to an EC access program as an actual parameter will not be changed while the program is executing. For example, when an array element is passed as an actual parameter to a program, if that program alters the value of the variables that determined the index, the results will be undefined.

(4) The protocol for passing parameters can implemented in a machine- independent fashion.

1.9. EC.SMPH

(1) The only operations on semaphores that need to be executed in a way that guarantees non-interference with other operations on semaphores are the following:

a. An operation that does not affect the counter value of the semaphore, but may put the process in the waiting state.

b. An operation to decrement the semaphore counter without any effect on the state of the process that executes it.

c. An operation to increment the semaphore counter that may put other processes in the active state.

1.10. EC.STATE

(1) The Extended Computer has at least three states: off, operating, and failed. Only the following transitions between states affect user programs:

- from off to operating
- from operating to failed.

(2) User programs cannot cause the transition into the operating state.

(3) A transition from operating to failed can either be caused by user programs or occur when malfunctions internal to the Extended Computer are detected. These internal malfunctions are other than those described in test programs contained in EC.PAR.1, EC.TIMER, EC.MEM, and EC.IO. It should be assumed that after this transition occurs, user programs will have at least a short interval to execute shut-down sequences before the computer stops operating. The minimum length of the interval before shut-down can be determined at system-generation time.

(4) Any actions that must be taken when a computer failure occurs are independent of the state of the user programs, and can be built into the EC.

1.11. EC.TIMER

(1) Avionics programs need timers that keep track of elapsed time, and that may signal when a given time interval has elapsed. They need to be able to set a timer to a starting value, start it, stop it, and read it whether it is running or not.

(2) The maximum timing capacity of a clock or a timer can be determined at system generation time.

(3) If a timer runs beyond a limit specified at run time, it should either halt or start over. Sometimes it should signal that a range limit has been reached.

(4) The worst acceptable error rate for all timers can be determined by users at system generation time. This error can be specified as a fraction of the running time, plus a constant error independent of the running time..

(5) Any number of timers can be implemented, provided that the number is known at system-generation time. There is no need to create or delete timers at run time.

(6) There are diagnostic programs that can test the hardware timers and the interrupt mechanism separately, but may interfere with proper execution of other programs.

ASSUMPTIONS ABOUT UNDESIRED EVENTS

2.1. EC.DATA

(1) User programs will not divide by zero.

(2) The result of any operation will not be outside the range of the destination variable.

(3) In a replace operation, user programs will not specify positions that do not appear within bitstrings or specify a substring with a start position that is higher than the stop position.

(4) After converting a numeric value to a bitstring, there will be no bits to the left of the most significant bit of the destination bitstring.

(5) Users will not supply a parameter in an array reference that is not in the index set of the array.

(6) Users will not fail to initialize a variable before its value is used in an operation.

(7) Users will not provide a range assertion about a variable that turns out to be false.

2.2. EC.IO

(1) User programs will not attempt to use an enabled read-write input data item as a !!destination!! or use an enabled read-write output data item as a !!source!!.

(2) User programs will not disable (enable) a data item that is already disabled (enabled).

2.3. EC.MEM

None.

2.4. EC.PAR.1

(1) Demand processes will not need to run so often as to cause a periodic process to miss its deadline.

(2) A periodic process will have a period greater than its deadline.

(3) A process will not be given a deadline that is shorter than the time it takes to execute.

2.5. EC.PAR.2

None.

2.6. EC.PGM.1

(1) Every program exit that will be chosen during execution will be connected to a succeeding command.

2.7. EC.PGM.2

(1) A user will not fail to assign a value to a built-in EC program variable.

2.8. EC.PGM.3

(1) A program will not invoke itself.

2.9. EC.SMPH

(1) There is a range of values that will suffice for all semaphores, and will not be exceeded by user programs.

2.10. EC.STATE.3 None.

2.11. EC.TIMER None.

APPENDIX D

Unimplemented Extended Computer Features

Not all of the capabilities described in this document have been provided in the current version of the Extended Computer. A few facilities, which are not currently needed by the application program, have not been implemented. An attempt to use an absent facility will result in an undesired event in the development version. The unimplemented features are described below.

1. Unimplemented features applying to more than one EC submodule

1.1. Undesired event detection in the production EC

Undesired event: None

Current use:

In the production version of the Extended Computer, no undesired events will be checked for. It will be assumed that user programs will invoke the EC facilities correctly.

1.2. Pointers pointing to a timer

Where described: EC.DATA.2.4, EC.TIMER.

Undesired event: %%unimplemented timer ptr%%

Current use:

A pointer may not be made to point to a timer.

1.3. Some forms of varying-type entities

Where described: EC.DATA, EC.PGM.2, EC.SMPH, EC.TIMER

Undesired event: %%unimplemented binding%%

Current use:

In the ++DCL_TYPE_CLASS++, ++DCL_ENTITY++ and ++DCL_ARRAY++ programs, if the type_list has more than one element, they must all be spectypes of the real typeclass. No other kind of entity is allowed to change its attributes, and no entity can assume attributes of more than one typeclass.

1.4. Invoking non-constant programs in a periodic process

Where described: EC.PGM.2, EC.PAR.1

Undesired event: %%unimplemented non-constant pgm in p proc%%

Current use:

Only a program constant, or a constant pointer to a program constant, may be invoked as part of a periodic process.

1.5. Using variables to specify attributes

Where described: EC.DATA.3, EC.SMPH.3, EC.TIMER.3

Undesired event: %%unimplemented attribute via variables%%

Current use:

> To specify an attribute (as defined in EC.DATA.3), a timer-attribute (as defined in EC.TIMER.3), or a semaphore-attribute (as defined in EC.SMPH.3), literals or ascons must be used.

1.6. Detection of %uninitialized entity%

Where described: EC.DATA, EC.PGM.2

Undesired event: None

Current use:

> %uninitialized entity% will not be detected, even in the development version. If a user fails to assign a value to an entity or array element that was declared with initial value UNDEF before he uses it, the behavior of his program is undefined. There is a built-in program pointer by that name, as for other UEs, but it will never be invoked by the EC as a result of detecting the error condition associated with it.

1.7. Pointers to programs that have parameters or more than one entrance.

Where described: EC.DATA, EC.PGM.2

Undesired event: %%unimplemented pgm ptr%%

Current use:

> A pointer may only refer to user-defined programs with one entrance, or to parameterless EC access programs.

1.8. !!list!!s of operands to EC access programs.

Undesired event: %%unimplemented operand list%%

Current use:

> An !!actual parameter!! may not be a !!list!! of !!actual parameter!!s, as described in EC.DATA.2.4.3, unless the !!actual parameter!! is a !!destination!!. In that case, section EC.DATA.2.4.2 applies. %%list mismatch%% will not be detected.

2. EC.DATA Unimplemented Features

2.1. Specifying substrings of bitstrings with variables

Undesired event: %%unimplemented variable substring%%

Current use:

> In the bitstring +REPLC+ program, p2, p3, and p4 must be given by literals or ascons.

2.2. Specifying a bitstring shift length with a variable

Undesired event: %%unimplemented variable shift length%%

Current use:

> In the +SHIFT+ program, p2 must be given by a literal or an ascon.

2.3. Varying the EXACT_REP attribute for numeric types.

Undesired event: %%unimplemented varying EXACT_REP%%

Current use:

If a type is declared to be of more than one spectype, then the spectype_list is not allowed to contain both a spectype that has the EXACT_REP attribute and a spectype that does not.

2.4. Using the EXACT_REP attribute for some resolutions

Undesired event: %%unimplemented EXACT_REP resolution%%

Current use:

For a type to have the EXACT_REP attribute, its resolution must be an exact integer power of two (for reals) or two milliseconds (for timeints).

2.5. Checking parameter type when it is given by a pointer

Undesired event: None.

Current use:

If an !!actual parameter!! is given by naming a pointer to an entity, and that entity is not of the proper type as required by the program being invoked, the result will be unpredictable; no UE will be raised.

2.6. Checking for %constant destination% with pointers

Undesired event: None.

Current use:

If a !!destination!! is given by naming a pointer to an entity, and that entity is not a variable, the result will be unpredictable; %constant destination% will not be raised.

2.7. Giving !!user threshold!! as a variable

Undesired event: %%unimplemented variable threshold%%

Current use:

The !!user threshold!! parameter of the numeric comparison programs must be given by a literal or an ascon.

2.8. Detecting %wrong attributes%

Undesired event: None.

Current use:

This undesired event will not be detected. If a program executes such that the UE would be raised were it detected, the effect of that program shall be undefined. There is a built-in program pointer by that name, as for other UEs, but it will never be invoked by the EC as a result of detecting the associated error.

2.9. +B_REAL_BCD+

Undesired event: %%unimplemented pgm%%

Current use:

These programs may not be called.

2.10. Arbitrary length bitstrings in +R_BITS_BCD+

Undesired event: %%unimplemented +R_BITS_BCD+ bits length%%

Current use:

p1 of +R_BITS_BCD+ must have a length that is an exact integer multiple of four.

3. EC.IO Unimplemented Features

3.1. Enabling/disabling i/o data items

Undesired event: %%unimplemented enable/disable%%

Current use:

+DISABLE+ or +ENABLE+ may not be called. The UEs %already enabled% and %already disabled% are not detected, and there are no built-in EC program pointers for error handling by those names.

3.2. Using variable or loadcon to access //FLTREC// elements

Undesired event: %%unimplemented non-const fltrec access%%

Current use:

In an <EL index //FLTREC//> operand, the index must be given by a literal or ascon.

4. EC.PAR Unimplemented Features

4.1. Periodic processes with periods that vary at run-time

Undesired event: %%unimplemented variable period%%

Current use:

The !!period!! parameter in ++P_PROCESS++ must be given as an ascon or a literal.

4.2. Periodic processes with deadlines that vary at run-time

Undesired event: %%unimplemented variable deadline%%

Current use:

The !!deadline!! parameter in ++P_PROCESS++ or ++D_PROCESS++ must be given by an ascon or literal.

4.3. Regions in programs invoked through pointers

Undesired event: %%unimplemented region in ptr-invoked pgm%%

Current use:

If programs that are invoked through pointers contain regions, the programs must contain equivalent regions. Two regions are considered equivalent if they have the same exclusion relations. For scheduling purposes, two such programs are also assumed to have the same maximum running time.

5. EC.PGM Unimplemented Features

5.1. Certain multi-exit built-in EC access programs

Undesired event: %%unimplemented multi-exit EC access program%%

Current use:

Only EC access programs computing a single output parameter that is real or boolean have more than one exit as described in EC.PGM.1.2.1. Invocations of other EC access programs that compute a single result must be followed by a null !!exit connector!! or one consisting of a single port.

5.2. Invocations of user-defined programs in !!par program!!

Undesired event: %%unimplemented user pgm in PAR%%

Current use:

Only !!invocation!!s of EC access programs may occur in the call-list of a !!par program!!.

5.3. Program loadcons and variables

Undesired event: %%unimplemented pgm loadcon/variable%%

Current use:

++DCL_ENTITY++ or ++DCL_ARRAY++ may not be invoked with p2 containing a program spec-type if p3 = LOADCON or VAR.

APPENDIX E

Input/Output Data Item Names

The following table lists all data items available from the A-7E version of the Extended Computer, and tells whether each one is read-only (R), write-only (W), or read-write (RW). The spectype of each is formed by removing the / or // brackets, and suffixing *_type*. For example, the spectype of /AOA/ is AOA_type.

Input data items		**Output data items**	
Item name	**R/RW**	**Item name**	**R/RW**
/ACAIRB/	R	//ANTSLAVE//	W
/ADCFAIL/	R	//ASAZ//	RW
/AOA/	R	//ASEL//	RW
/ANTGOOD/	R	//ASLAZ//	RW
/ARPINT/	R	//ASLCOS//	RW
/ARPPAIRS/	R	//ASLEL//	RW
/ARPQUANT/	R	//ASLSIN//	RW
/BAROADC/	R	//AUTOCAL//	W
/BMBDRAG/	R	//AZRING//	RW
/BRGSTA/	R	//BAROHUD//	RW
/DIMWC/	R	//BMBREL//	W
/DGNDSP/	R	//BMBTON//	W
/DRFTANG/	R	//BRGDEST//	W
/DRSFUN/	R	//COMPCTR//	W
/DRSMEM/	R	//COMPFAIL//	W
/DRSREL/	R	//CURAZCOS//	RW
/ELECGOOD/	R	//CURAZSIN//	RW
/FLYTOTOG/	R	//CURENABL//	W
/FLYTOTW/	R	//CURPOS//	RW
/GUNSSEL/	R	//DESTPNT//	RW
/HUDREL/	R	//ENTLIT//	W
/IMSAUTOC/	R	//FIRRDY//	W
/IMSMODE/	R	//FLTDIRAZ//	RW
/IMSREDY/	R	//FLTREC//	W
/IMSREL/	R	//FPANGL//	W
(blank)	-	//FPMAZ//	RW
(blank)	-	//FPMEL//	RW
(blank)	-	//GNDTRK//	W
/KBDINT/	R	//GNDTRVEL//	W
/LOCKEDON/	R	//HUDAS//	RW
/MA/	R	//HUDASL//	RW
/MACH/	R	//HUDFPM//	RW
/MAGHCOS/	R	//HUDPUC//	RW
/MAGHSIN/	R	//HUDSCUE//	RW
/MFSW/	R	//HUDVEL//	RW
/MODEROT/	R	//HUDWARN//	RW
/MULTRACK/	R	//IMSNA//	W
/PCHCOS/	R	//IMSSCAL//	W
/PCHSIN/	R	//KELIT//	W

Input data items		Output data items	
Item name	R/RW	Item name	R/RW
/PMDCTR/	R	//LATGT70//	W
/PMHOLD/	R	//LFTDIG//	W
/PMNORUP/	R	//LLITDEC//	W
/PMSCAL/	R	//LLITE//	W
/PMSLAND/	R	//LLIT322//	W
/PNLTEST/	R	//LLITW//	W
/PRESPOS/	R	//LSOLCUAZ//	RW
/RADALT/	R	//LSOLCUEL//	RW
/RE/	R	//LWDIG1//	W
/RNGSTA/	R	//LWDIG2//	W
/ROLLCOSI/	R	//LWDIG3//	W
/ROLLSINI/	R	//LWDIG4//	W
/SINEVEL/	R	//LWDIG5//	W
/SINHDG/	R	//LWDIG6//	W
/SINLAT/	R	//LWDIG7//	W
/SINLONG/	R	//MAGHDGH//	RW
/SINNVEL/	R	//MAPOR//	RW
/SINPTH/	R	//MARKWIN//	RW
/SINROL/	R	//PTCHANG//	RW
/SLEWRL/	R	//PUACAZ//	RW
/SLEWUD/	R	//PUACEL//	RW
/SLTRNG/	R	//RNGHND//	W
/STA1RDY/	R	//RNGTEN//	W
/STA2RDY	R	//RNGUNIT//	W
/STA3RDY/	R	//ROLLCOSH//	RW
/STA6RDY/	R	//ROLLSINH//	RW
/STA7RDY/	R	//STEERAZ//	W
/STA8RDY/	R	//STEEREL//	W
/TD/	R	//STERROR//	W
/TAS/	R	//TSTADCFLR//	W
/THDGCOS/	R	//ULITN//	W
/THDGSIN/	R	//ULITS//	W
/UPDATTW/	R	//ULIT222//	W
/WAYLAT/	R	//ULIT321//	W
/WAYLON/	R	//USOLCUAZ//	RW
/WAYNUM1/	R	//USOLCUEL//	RW
/WAYNUM2/	R	//UWDIG1//	W
/WEAPTYP/	R	//UWDIG2//	W
/XGYCNT/	R	//UWDIG3//	W
/XVEL/	R	//UWDIG4//	W
/YGYCNT/	R	//UWDIG5//	W
/YVEL/	R	//UWDIG6//	W
/ZGYCNT/	R	//VERTVEL//	RW
/ZVEL/	R	//VTVELAC//	RW
		//XCOMMC//	RW
		//XCOMMF//	RW
		//XGYCOM//	W
		//XSLEW//	W
		//XSLSEN//	W
		//YCOMM//	RW
		//YGYCOM//	W
		//YSLEW//	W
		//YSLSEN//	W
		//ZGYCOM//	W

Input data items		Output data items	
Item name	**R/RW**	**Item name**	**R/RW**
		//ZSLEW//	W
		//ZSLSEN//	W

Note: //FLTREC// is an array; the number of elements is given by the integer system generation parameter #nbr fltrec elements#.

The following data items have events (signalled by incrementing a semaphore) associated with them:

Event	Semaphore	Region
@T(!+/ENTERSW/ occurred+!)	ENTSWSEM	ENTSWMEM_reg
@T(!+/KBDENBL/ occurred+!)	ENBLSEM	ENBLSEM_reg
@T(!+/MARKSW/ occurred+!)	MARKSEM	MARKSEM_reg
@T(!+/KBDINT/ ready+!)	KBINTSEM	KBINTSEM_reg

APPENDIX F

Data Representation Catalogue

For some type classes, the Extended Computer is capable of providing more than one kind of representation. The version has no effect on the outcome of an EC operation, but some versions allow some operations to be performed more efficiently than other versions.

The following table lists the provided version names for each EC type class. When declaring a specific type, users request a particular version by using these names.

Typeclass	Specific type	Version names	Version properties
BITS		Any	B1
PGM	Any	PRO1	N/A
PTR		Any	P1
REAL	Any	R1	N/A
SEMAPHORE	Any	S1	N/A
TIMEINT	Any	T1	N/A
TIMER	Any	C1	N/A

APPENDIX G

Extended Computer Efficiency Guide

1. Introduction

This appendix provides information that will allow programmers of the Extended Computer [EC] write programs with improved run-time performance, where *performance* is defined as an unspecified function of the execution time and memory space required by the program. In this appendix, the words *bigger* and *smaller*, *slower* and *faster* are adjectives describing the memory space and execution time, respectively, of programs. If one of two programs uses less space than the other and runs no slower it is considered more efficient. Similarly, if one of two programs runs faster than the other without using more space, it is considered more efficient. In all other cases, any judgement about efficiency depends upon the availability of the various resources.

The appendix is arranged in sections that correspond to chapters and sections in this document. The performance information is usually not given in quantitative terms; rather, we describe alternative programming situations and describe qualitatively the advantages and disadvantages of each. Where quantitative data is given, actual numbers are represented by symbolic names, with their values given at the end of the appendix. S and T are used to denote cost/saving of space and time, respectively, when only one is involved.

In this appendix, *length* refers to the !!version *n* characteristic!! of the type (see EC.DATA). In this appendix, *ascon* is taken to include both literals and assembly-time constants.

Following the guidelines of this appendix will produce a more efficient EC program. Should the implementation of the EC change (e.g., a new host computer is acquired, or different storage schemes adopted) the program will still be correct, although perhaps not as efficient.

The following assumptions are made:

(a) the reader is familiar with the programmer's interface to the Extended Computer presented in this document, including the unimplemented features specified in Appendix D;

(b) the programmer will not modify a program to gain efficiency if that modification results in the program that no longer meets its requirements; for instance, he will not provide a process !!deadline!! greater than that which his requirements allow, even if doing so will enhance his program's efficiency.

USING THIS FEATURE MAY INCREASE EFFICIENCY	USING THIS FEATURE MAY DECREASE EFFICIENCY
All Sections	
Giving all input actual parameters to a program as ascons rather than loadcons or variables	
For all programs that take parameters of version *n*, giving those parameters as entities whose !!version *n* characteristic!! is small, especially ##entity length## or less.	
EC.DATA.1.3 -- Scalar literals	

USING THIS FEATURE MAY INCREASE EFFICIENCY	USING THIS FEATURE MAY DECREASE EFFICIENCY
Numeric literals of less precision	
Bitstring literals of smaller !!version 1 characteristic!!, especially ##entity length## or less.	
EC.DATA.2.0 -- Use of of data sets	
The most efficiency from ranking data sets will result if all entities used within a particular code segment belong to data sets that are ranked highest within that code segment.	
EC.DATA.2.1 -- Declaration of specific types	
None.	None.
EC.DATA.2.3 -- Declaration of entities and arrays	
	Declaring entities or arrays of numeric types whose ratio of range to resolution is high (S)
	Declaring entities or arrays of bitstring types whose length is high (S)
Declaring arrays with indexset lower bound equal to 0	Declaring arrays with indexset lower bound not zero
EC.DATA.2.4 -- Operand descriptions	
Specifying subrange information for operations as often and as tightly as possible. Exceptions: range assertions need not appear for constant operands; range assertions need not appear for both source and destination(s) in a +SET+ operation; range assertions need not appear for more than one destination in a list.	Using a pointer to point to an ASCON will negate any efficiency gained by using an ASCON instead of a LOADCON
Using destination lists instead of serial assignment	
Using the NOSTORE operand form wherever possible	
Giving numeric literals as !!typed literal!!s, when the spectype named is one that will allow other efficiency gains (e.g., making the source and destination spectype the same for transfer and numeric	

USING THIS FEATURE MAY INCREASE EFFICIENCY	**USING THIS FEATURE MAY DECREASE EFFICIENCY**
operations)	
EC.DATA.2.5 -- Transfer operations	
Giving !!source!! and !!destination!! of same specific type	
See also ##operation cost##.	
EC.DATA.2.6.1 -- Numeric comparison operations	
Giving !!user threshold!! of 0	Giving !!source!!'s of different specific types
Giving one !!source!! as an ascon, if its value is zero	
Giving both !!source!!s and the !!destination!! of the same specific type	
See also ##operation cost##.	
EC.DATA.2.6.2 -- Numeric calculations	
Making one !!source!! of +ADD+ or +SUB+ an ascon, if its value is zero	For +ADD+ and +SUB+, giving !!source!!'s of different specific types
Making both !!source!! of +ADD+ and +MUL+ the same specific type.	
Making the !!destination!! of +ADD+ or +SUB+ the same specific type as both of the !!source!!'s	
Making one !!source!! of +MUL+ an ascon if its value is one, or a power of two	
Making p2 of +DIV+ an ascon, if its value is one or a power of two	Making p3 of +DIV+ $true$
Giving p1 of +SIGN+ as an ascon	
See also ##operation cost##.	
EC.DATA.2.6.3 -- Conversions to reals	
Giving !!radix pt ident!! as an ascon	
Converting a bitstring whose length is ##entity length## or 2 × ##entity length##	
See also ##operation cost##.	
EC.DATA.2.6.4 -- Conversions to timeints	

USING THIS FEATURE MAY INCREASE EFFICIENCY	USING THIS FEATURE MAY DECREASE EFFICIENCY
See ##operation cost##.	
EC.DATA.2.7.1 -- Bitstring comparisons	
See ##operation cost##.	
EC.DATA.2.7.2 -- Bitstring calculations	
See ##operation cost##.	Using +REPLC+ to insert a substring into a bitstring when the value of the substring is known at sysgen time. Better to use +AND+ with a mask.
EC.DATA.2.7.3 -- Conversions to bitstring	
Giving !!radix pt ident!! as an ascon See also ##operation cost##.	
EC.DATA.2.8 -- Pointer operations	
	None.
EC.IO -- Input/Output operations	
Using an already-disabled i/o item for storage in place of declaring entities (S). This savings may be lost if the entity would have been declared to be in a data set that is highly ranked where the i/o item is used.	Disabling an i/o item merely to avoid declaring one entity. Enabling and disabling entities are expensive enough so that several entities must be saved before a net gain is realized.
	Using one of the *x*_success builtin booleans, even if the statement in which it appears is never executed. (S)
Reading and writing i/o items within a !!par program!! if those items are in the same Data Item Class ([REQ], Section 2.5) except for the SIR-DOW3 class. (T) Writing to the following groups of output items within a !!par program!!: //HUDAS// //HUDASL// //HUDFPM// //HUDPUC// //HUDSCUE// //HUDVEL// //HUDWARN// (T)	
EC.MEM -- Memory module	
	None.

USING THIS FEATURE MAY INCREASE EFFICIENCY	**USING THIS FEATURE MAY DECREASE EFFICIENCY**
EC.PAR.1 -- Process mechanisms	
Making !!deadline!! and !!period!! as large as possible	
Giving !!on/off!! as an ascon, rather than a variable whose value does not change	Having !!on/off!! for a process set to true longer than necessary (T)
EC.PAR.2 -- Exclusion regions	
	Having more ordered pairs in the exclusion relation than is required.
EC.PGM.1 -- Program construction	
Using an !!exit connector!! in which the exits are contiguous evenly-spaced values in the domain of the computed result of the program. Exits that correspond to contiguous integers are especially efficient.	
Using the null form of the !!exit connector!!	
Using ++PAR++ to group input or output operations when the items to be read or written all inhabit the same TC-2 word (specified in [REQ], Chapter 2).	
EC.PGM.2 -- Program entities	
Maximum efficiency will result if the most often invoked programs are ranked higher.	
EC.PGM.3 -- Program invocation facilities	
	None.
EC.SMPH -- Synchronization variables and operations	
	None.
EC.STATE -- Extended computer state	
	None.
EC.TIMER -- Timer facilities	
	Declaring a timer with the WRAP attribute

USING THIS FEATURE MAY INCREASE EFFICIENCY	**USING THIS FEATURE MAY DECREASE EFFICIENCY**
	Letting timers with WRAP attribute reach an upper or lower limit (T)

##term## definitions

##entity length## 16

##operation cost## Given in the table below. Taking the fastest possible EC addition as a basic unit of cost, the following operations have these relative costs. The numbers given are a range, or a minimum.

Operation	Approximate cost relative to addition
+SUB+	1
+MUL+	4
+DIV+	4-5
numeric comparisons	1-2
+ABSV+	1-2
+COMPLE+	1-2
+SIGN+	1
time-real conversions	0-5
real-time conversions	0-5
bitstring comparisons	1
bitstring calculations	1
+SHIFT+	1-2
+REPLC+	
real-bits conversions	0-5
bits-real conversions	0-5

EC.INDEX: Indices to the Document

1. Access programs

2. Local type definitions

In addition, Appendix E lists a set of builtin bitstring spectypes.

3. Dictionary terms

4. Undesired events

5. System generation parameters

6. Built-in objects

7. Events signalled by incrementing a semaphore

In addition, users may request timer-related events by supplying their own semaphores. See EC.TIMER

8. Reserved words

References

[ADP] Brooks, Iverson; *Automatic Data Processing, System/360 Edition,* Wiley, 1969.

[ADT] Clements, Faulk, Parnas; *Interface Specifications for the SCR (A-7E) Application Data Types Module,* NRL Report 8734, 23 August 1983.

[APC] Faulk, S.; ‘‘Pseudo-Code Language for the A-7E OFP’’, internal memorandum, April 1982

[BELP73] Belpaire, Wilmotte; ‘‘A Semantic Approach to the Theory of Parallel Processes’’; in International Computing Symposium 1973.

[DIJK68] Dijkstra, E.; ‘‘Cooperating Sequential Processes’’, in *Programming Languages,* ed. F. Genuys; Academic Press, 1968; pp.43-112.

[DIM] Parker, Heninger, Parnas, Shore; *Abstract Interface Specifications for the A-7E Device Interface Module,* NRL Memorandum Report 4385, November, 1980.

[REED79] Reed, Kanodia; ‘‘Synchronization with Eventcounts and Sequencers’’; *Comm. ACM*, v. 22, no. 2 (1979).

[REQ] Heninger, Kallander, Parnas, Shore; *Software Requirements for the A-7E Aircraft*, NRL Memorandum Report 3876; Nov 1978.

[SO] Clements, Parker, Parnas, Shore, Britton; *A Standard Organization for Specifying Abstract Interfaces*, NRL Report 8815, June 1984.

[TRACE] Parnas, "Trace Specifications for D-Operations", NRL Technical Memorandum 7590-000:DP, to be published.

[TT] Alspaugh, Clements, Mullen, Parnas, Weiss; ‘‘Interface Specifications for the SCR (A-7) Translator Tools Module’’, NRL Memorandum Report in preparation, draft 22 October 1984.

[WUER76] Wuerges, Parnas; ‘‘Response to Undesired Events in Software Systems’’; *Proc. 2nd Intl. Conf. Softare Eng.*, pp. 437-446; 1976.

Acknowledgements

The authors gratefully acknowledge the hard work and careful reviews provided by the following people:

Naval Weapons Center, China Lake, CA:

Jack Basden
Richard Fryer
Sandra Fryer
Dawn Janney
Ray Martinusen
Jo Miller
Lee Thomson
Robert Westbrook
Richard Wolff
Janice Zenor

Vought Corporation, Dallas, TX:

Glenn Cooper
Dwight Hill

USAF A-7D/K OFP Detachment, Tucson, AZ:

Mark Jacobson
Richard Breisch

Bell Telephone Laboratories, Columbus, OH:

Don Utter

Grumman Aerospace Corp., Bethpage, NY:

Stephanie White

Computer Science and Systems Branch, Naval Research Laboratory,Washington, DC:

Tom Alspaugh
Stuart Faulk
Bruce Labaw
Larry Morell
Dr. John Shore

In addition, special thanks go to Preston Mullen, who contributed significantly to the design of several of the interfaces contained in this document.

(end)

Table of Contents